第二章

用户和用户组管理

用户管理常用命令

用户账号添加命令useradd或adduser

修改用户命令usermod

删除用户命令userdel

用户口令管理命令passwd

Linux用户三类：root用户、虚拟用户（bin、deamon/adm/ftp等）、普通真实用户

用户组管理常用命令

groupadd可指定用户组名称来建立新的用户组，需要时可从系统中去的新用户组值

groupadd [option] [groupname]

groupmod可指定用户组名称来修改新的用户组号或用户组名称

groupmod [option] [groupname]

文件和目录操作

文件操作常用命令

cp 命令可以将给出的文件或目录复制到另一个文件或目录中

cp [option] [source] [destination]

cp –a 使拷贝的文件权限和修改日期不发生变化

cp –r 可拷贝整个目录

mv命令可用于将文件或目录从一个位置移动到另一个位置

mv [option] [source] [destination]

经常使用mv命令对文件进行重命名

rm命令提供删除文件功能，该命令可以删除目录中的一个或多个文件或子目录

rm [option] [filename or directoryname]

rm命令的-r和-f选项经常使用

选项	意义
-f	忽略不存在的文件，从不给出提示
-r	指示rm将参数中列出的全部目录和子目录均递归地删除
-i	进行交互式删除

mkdir创建目录

mkdir [option] [directoryname]

rmdir命令可以删除一个或多个目录，在删除目录时，目录必须为空

rmdir [option] [directoryname]

rmdir –p递归删除目录

目录切换命令cd

文件和目录权限管理

chmod 命令的格式为：

chmod [userType] [signal] [type] [filename]

chown命令可改变文件或命令的属主（**root**使用）

chown [option] [owner] [filename]

-R是最常用的选项，对目前目录下的所有文件与子目录进行相同的拥有者变更

find命令的基本格式

find [路径] [选项] [操作]

路径是find命令所查找的目录路径，例如用.来表示当前目录，用/来表示系统根目录

选项用于指定查找条件，如：可以指定按照文件属主、更改时间、文件类型等条件来查找

操作用于指定结果的输出方式

选项	意义
name	根据文件名查找文件
perm	根据文件权限查找文件
prune	使用这一选项可以使find命令不在当前指定的目录中查找，如果同时使用-depth选项，那么-prune将被find命令忽略
user	根据文件属主查找文件
group	根据文件所属的用户组查找文件
mtime -n +n	根据文件的更改时间查找文件，-n表示文件更改时间距今在n天之内，+n表示文件更改时间距今在n天前
nogroup	查找无有效所属组的文件，即该文件所属的组在/etc/groups中不存在
nouser	查找无有效属主的文件，即该文件的属主在/etc/passwd中不存在
-newer file1 ! file2	查找更改时间比文件file1新但比文件file2旧的文件
type	查找某一类型的文件，type后跟的子选项及其意义如下： b:块设备文件 d:目录 c:字符设备文件 p:管道文件 l:符号链接文件 f:普通文件
size n:[c]	查找文件长度为n块的文件，带有c时表示文件长度以字节计
depth	在查找文件时，首先查找当前目录中的文件，然后再在其子目录中查找

操作名称	意义
print	将匹配的文件输出到标准输出
exec	对匹配的文件执行该参数所给出的Shell命令。相应命令的形式为'command' { } \;，注意{ }和\；之间的空格
ok	和-exec的作用相同，只不过以一种更为安全的模式来执行该参数所给出的Shell命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行

Vi编辑器

首先按下斜杠按钮（/），光标会自动移到vi编辑器下方的命令行，用户输入待搜索的字符串，按下Enter键开始搜索，vi编辑器可能用三种方式响应用户的搜索

vi编辑器的替换命令的基本格式为：

:s/old_string/new_string

:s/old_string/new_string/g

第三章 正则表达式

正则表达式基础

正则表达式是由一串字符和元字符构成的字符串，简称RE（Regular Expression）。

正则表达式的主要功能是文本查询和字符串操作，正则表达式可以匹配文本的一个字符或字符集合。

符号	意义
*	0个或多个在*字符之前那个普通字符
.	匹配任意字符
^	匹配行首，或后面字符的非
$	匹配行尾
[]	匹配字符集合
\	转义符，屏蔽一个元字符的特殊意义
\<\>	精确匹配符号
\{n\}	匹配前面字符出现n次
\{n,\}	匹配前面字符至少出现n次
\{n,m\}	匹配前面字符出现n次与m次之间

***符号用于匹配前面一个普通字符的0次或多次重复**

hel*o： *符号前面的普通字符是l，*字符就表示匹配l字符0次或多次，如字符串helo、hello、hellllllo都可以由hel*o来表示

.符号用于匹配任意一个字符

…73.表示前面三个字符为任意字符，第4和第5个字符是7和3，最后一个字符为任意字符，如xcb738、4J973U都能匹配上述字符串

^符号用于匹配行首，表示行首的字符是^字符后面的那个字符

^cloud表示匹配以cloud开头的行

**$符号匹配行尾**，$符号放在匹配字符之后a、b、1、2等字符属于普通字符，普通字符可以按照字面意思理解，如：a只能理解为英文的小写字母a，没有其他隐藏含义。

micky$表示匹配以micky结尾的所有行

^$ 表示空白行

[]匹配字符集合，该符号支持穷举方法列出字符集合的所有元素

[0123456789]、[0-9]……

[A-Za-z] [A-Za-z]* 匹配所有英文单词

\符号是转义符，用于屏蔽一个元字符的特殊意义

\. \$ \^

\<\>符号是精确匹配符号，该符号利用\符号屏蔽<>符号

\<the\> 精确匹配单词the，them, they等不匹配

\{\}系列符号表示前一个字符的重复

\{n\}：匹配前面字符出现n次 ，如 JO\{3\}B 匹配JOOOB

\{n,\}：匹配前面字符至少出现n次，如JO\{3,\}B匹配JOOOB、JOOOOB、JOOOOOB等字符串

\{n,m\}：匹配前面字符出现n次与m次之间，如JO\{3,6\}B匹配JOOOB、JOOOOOOB等字符串

[a-z] \{5\}： 匹配5个小写英文字母，比如hello、house等

正则表达式扩展

awk和perl等linux工具还支持正则表达式扩展出来的一些元字符，这些元字符如下表所示

符号	意义
?	匹配0个或1个在其之前的那个普通字符
+	匹配1个或多个在其之前的那个普通字符
()	表示一个字符集合或用在expr中
|	表示“或”意义，匹配一组可选的字符

通配

通配（globbing）是把一个包含通配符的非具体文件名扩展到存储在计算机、服务器或者网络上的一批具体文件名的过程

最常用的通配符包括正则表达式元字符：?、*、[]、{}、^等，通配符与元字符意义不完全相同：

*符号不再表示其前面字符的重复，而是表示任意位的任意字符

?字符表示一位的任意字符

^符号在通配中不代表行首，而是代表取反意义

grep命令

grep [选项][模式][文件...]

模式可以是字符串，也可以是变量，还可以是正则表达式。需要说明的是，无论模式是何种形式，只要模式中包含空格，就需要使用双引号将模式括起来

文件可以有多个，亦可以用通配来表示

选项	意义
-c	只输出匹配行的数量
-i	搜索时忽略大小写
-h	查询多文件时不显示文件名
-l	只列出符合匹配的文件名，而不列出具体匹配行
-n	列出所有匹配行，并显示行号
-s	不显示不存在或无匹配文本的错误信息
-v	显示不包含匹配文本的所有行
-w	匹配整词
-r	递归搜索，不仅搜索当前工作目录，而且搜索子目录
-E	支持扩展的正则表达式
-F	不支持正则表达式，按照字符串的字面意思进行匹配

第四章 sed命令和awk编程（上）

sed基本用法

sed（**s**tream **ed**itor）是流编辑器，可对文本文件和标准输入进行编辑。sed只是对缓冲区中原始文件的副本进行编辑，并不编辑原始的文件，如果需要保存改动内容，可以选择使用下面两种方法：重定向和w编辑命令

调用sed的三种方法

在Shell命令行输入命令调用sed，格式为：

sed [选项] ‘sed命令’ 输入文件

将sed命令插入脚本文件后，然后通过sed命令调用它，格式为：

sed [选项] -f sed脚本文件 输入文件

将sed命令插入脚本文件后，最常用的方法是设置该脚本文件为可执行，然后直接执行该脚本文件，格式为：

./sed脚本文件 输入文件

第二种方法脚本文件的首行不以**#!/bin/sed –f**开头；第三种方法脚本文件的首行是**#!/bin/sed –f**

Sed命令选项

选项	意义
-n	不打印所有行到标准输出
-e	表示将下一个字符串解析为sed编辑命令，如果只传递一个编辑命令给sed，-e选项可以省略
-f	表示正在调用sed脚本文件

sed文本定位方法

选项	意义
x	x为指定行号
x,y	指定从x到y的行号范围
/pattern/	查询包含模式的行
/pattern/pattern/	查询包含两个模式的行
/pattern/,x	从与pattern的匹配行到x号行之间的行
x,/pattern/	从x号行到与pattern的匹配行之间的行
x,y!	查询不包括x和y行号的行

sed编辑命令

选项	意义
p	打印匹配行
=	打印文件行号
a\	在定位行号之后追加文本信息
i\	在定位行号之前插入文本信息
d	删除定位行
c\	用新文本替换定位文本
s	使用替换模式替换相应模式
r	从另一个文件中读文本
w	将文本写入到一个文件
y	变换字符
q	第一个模式匹配完成后退出
l	显示与八进制ASCII代码等价的控制字符
{}	在定位行执行的命令组
n	读取下一个输入行，用下一个命令处理新的行
h	将模式缓冲区文本拷贝到保持缓冲区
H	将模式缓冲区文本追加到保持缓冲区
x	互换模式缓冲区和保持缓冲区内容
g	将保持缓冲区内容拷贝到模式缓冲区
G	将保持缓冲区内容追加到模式缓冲区

sed基本编辑

追加文本：匹配行后面插入

插入文本：匹配行前面插入

修改文本：将所匹配的文本行利用新文本替代

删除文本：将指定行或指定行范围进行删除

sed替换文本操作将所匹配的文本行利用新文本替换，替换文本与修改文本功能有相似之处，它们之间的区别在于：替换文本可以替换一个字符串，而修改文本是对整行进行修改

替换文本的格式为：

s/被替换的字符串/新字符串/[替换选项]

选项	意义
g	表示替换文本中所有出现被替换字符串之处
p	与-n选项结合，只打印替换行
w 文件名	表示将输出定向到一个文件

默认情况下，sed s命令将替换后的全部文本都输出，如果要求只打印替换行，需要结合使用-n和p选项，命令格式如下：

sed -n ‘s/被替换的字符串/新字符串/p’ 输入文件

g选项是替换文本中所有出现被替换字符串之处

w选项后加文件名表示将输出定向到这个文件，与sed编辑命令中的w是不矛盾的

从文件中读入文本，r选项

退出命令： q选项表示完成指定地址的匹配后立即退出

变换命令： y选项表示字符变换，它将一系列的字符变换为相应的字符

sed ‘y/被变换的字符序列/变换的字符序列/’ 输入文件

sed y命令要求被变换的字符序列和变换的字符序列等长，否则sed y命令将报错

sed编辑命令中的**{}符号**可以指定在定位行上所执行的命令组，它的作用与sed的-e选项类似，都是为了在定位行执行多个编辑命令

第四章 sed命令和awk编程（下）

awk基本用法

awk编程模型

调用awk有三种方法（与sed类似）

在Shell命令行输入命令调用awk，格式为

awk [-F 域分隔符] ‘awk程序段’ 输入文件

将awk程序段插入脚本文件后，然后通过awk命令调用它：

awk -f awk脚本文件 输入文件

将awk命令插入脚本文件后，最常用的方法是设置该脚本文件为可执行，然后直接执行该脚本文件，格式为：

./awk脚本文件 输入文件

第二种方法脚本文件的首行不以#!/bin/awk –f开头；第三种方法脚本文件的首行是#!/bin/awk –f

awk记录和域

awk认为输入文件是结构化的，awk将每个输入文件行定义为记录，行中的每个字符串定义为域，域之间用空格、Tab键或其他符号进行分割，分割域的符号就叫分隔符

awk改变域分隔符有两种方法：-F 选项；FS变量

awk关系、布尔运算符、表达式

运算符	意义
<	小于
>	大于
<=	小于等于
>=	大于等于
==	等于
!=	不等于
~	匹配正则表达式
!~	不匹配正则表达式
||	逻辑或
&&	逻辑与
!	逻辑非

与其他编程语言一样，awk表达式用于存储、操作和获取数据，一个awk表达式可由数值、字符常量、变量、操作符、函数和正则表达式自由组合而成

变量是一个值的标识符，定义awk变量非常方便，只需定义一个变量名并将值赋给它即可。变量名只能包含字母、数字和下划线，而且不能以数字开头

运算符	意义
+	加
-	减
*	乘
/	除
%	模
^或**	乘方
++x	在返回x值之前，x变量加1
x++	在返回x值之后，x变量加1

计算stureocrd中的空白行（^首行 $尾行）

awk系统变量

系统变量可分为两种：

第1种用于改变awk的缺省值，如域分隔符；

第2种用于定义系统值，在处理文本时可以读取这些系统值，如记录中的域数量、当前记录数、当前文件名等，awk动态改变第2种系统变量的值

变量名	意义
$n	当前记录的第n个域，域间由FS分割
$0	记录的所有域
ARGC	命令行参数的数量
ARGIND	命令行中当前文件的位置（以0开始标号）
ARGV	命令行参数的数组
CONVFMT	数字转换格式
ENVIRON	环境变量关联数组
ERRNO	最后一个系统错误的描述
FIELDWIDTHS	字段宽度列表，以空格键分割
FILENAME	当前文件名
FNR	浏览文件的记录数
FS	字段分隔符，默认是空格键
IGNORECASE	布尔变量，如果为真，则进行忽略大小写的匹配
NF	当前记录中的域数量
NR	当前记录数
OFMT	数字的输出格式
OFS	输出域分隔符，默认是空格键
ORS	输出记录分隔符，默认是换行符
RLENGTH	由match函数所匹配的字符串长度
RS	记录分隔符，默认是空格键
RSTART	由match函数所匹配的字符串的第1个位置
SUBSEP	数组下标分隔符，默认值是\034

awk格式化输出

awk的一大主要功能是产生报表，报表就要求按照预定的格式输出，awk借鉴C语言的语法，定义了printf输出语句，它可以规定输出的格式。**printf的基本语法如下：**

printf (格式控制符,参数)

格式控制符分为awk修饰符和格式符

修饰符	意义
-	左对齐
width	域的步长
.prec	小数点右边的位数

运算符	意义
%c	ASCII字符
%d	整型数
%e	浮点数，科学记数法
%f	浮点数
%o	八进制数
%s	字符串
%x	十六进制数

awk内置字符串函数

函数名	意义
gsub(r,s)	在输入文件中用s替换r
gsub(r,s,t)	在t中用s替换r
index(s,t)	返回s中字符串第一个t的位置
length(s)	返回s的长度
match(s,t)	测试s是否包含匹配t的字符串
split(r,s,t)	在t上将r分成序列s
sub(r,s,t)	将t中第1次出现的r替换为s
substr(r,s)	返回字符串r中从s开始的后缀部分
substr(r,s,t)	返回字符串r中从s开始长度为t的后缀部分

4个字符

awk条件语句和循环语句

awk条件语句和循环语句与**C**语言的语法完全一样

if (条件表达式)
    动作1
[else
    动作2]
while (条件表达式)
    动作
do
    动作
while (条件表达式)
for (设置计数器初值;测试计数器;计数器变化)
    动作

awk数组

数组是用于存储一系列值的变量，这些值之间通常是有联系的，可通过索引来访问数组的值，索引需要用中括号括起，

数组的基本格式为：

array[index]=value

关联数组是指数组的索引可以是字符串，也可以是数字

关联数组在索引和数组元素值之间建立起关联，对每一个数组元素，awk自动维护了一对值：索引和数组元素值

关联数组的值无需以连续的地址进行存储，awk的所有数组都是关联数组

字符串和数字之间的差别是明显的，如，我们使用array[09]指定一个数组值，如果换成array[9]就不能指定到与array[09]相同的值

split(r,s,t)函数将字符串以t为分隔符，将r字符串拆分为字符串数组，并存放在s中，此时s通常就是一个数组

awk 'BEGIN {print split("abc/def/xyz",str,"/")}'　返回3

上面命令以“/”为分隔符，将字符串abc/def/xyz分开，并存在str数组中，split函数的返回值是数组的大小

awk可使用for循环打印数组内容

for (variable in array)

do something with array[variable]

ARGC是ARGV数组中元素的个数，与C语言一样，从ARGV[0]开始，到ARGV[ARGC-1]结束

ENVIRON变量存储了Linux操作系统的环境变量

argv[0]存储的是程序名

第五章 文件的合并、排序和分割

sort命令

Linux的sort命令就是一种对文件排序的工具，sort命令的功能十分强大，是Shell脚本编程时常用的文件排序工具；

sort命令与awk一样，将文件看作记录和域进行处理，默认的域分隔符是空格符，sort命令的格式为：

sort [选项] [输入文件]

选项	意义
-c	测试文件是否已经被排序
-k	指定排序的域
-m	合并两个已排序的文件
-n	根据数字大小进行排序
-o [输出文件]	将输出写到指定的文件，相当于将输出重定向到指定文件
-r	将排序结果逆向
-t	改变域分隔符
-u	去除结果中的重复行

不是按数字大小排

uniq命令

uniq命令用于去除文本文件中的重复行，这类似于sort命令的-u选项

sort -u命令时，所有重复记录都被去掉

uniq命令去除的重复行必须是连续重复出现的行，中间不能夹杂任何其他文本行

选项	意义
-c	打印每行在文本中重复出现的次数
-d	只显示有重复的记录，每个重复记录只出现一次
-u	只显示没有重复的记录

uniq命令的-c选项打印每行在文本中重复出现的次数，常用于计数功能，-c选项是uniq最有用的选项

uniq命令的-d和-u选项正好相反，-d选项用于显示有重复的记录，而-u选项显示没有重复的记录

join命令

join命令用于实现两个文件中记录的连接操作，连接操作将两个文件中具有相同域的记录选择出来，再将这些记录所有的域放到一行（包含来自两个文件的所有域）

join [选项] 文件1 文件2

选项	意义
-a1或-a2	除了显示以共同域进行连接的结果外，-a1表示还显示第1个文件中没有共同域的记录，-a2则表示显示第2个文件中没有共同域的记录
-i	比较域内容时，忽略大小写差异
-o	设置结果显示的格式
-t	改变域分隔符
-v1或-v2	跟-a选项类似，但是，不显示以共同域进行连接的结果
-1和-2	-1用于设置文件1用于连接的域，-2用于设置文件2用于连接的域

当两个文件进行连接时，文件1中的记录可能在文件2中找不到共同域，反过来，文件2中也可能存在这样的记录，join命令的结果默认是不显示这些未进行连接的记录的

-a和-v选项就是用于显示这些未进行连接的记录，-a1和-v1指显示文件1中的未连接记录，而-a2和-v2指显示文件2中的未连接记录

-a和-v选项的区别在于：-a选项显示以共同域进行连接的结果，而-v选项则不显示这些记录

join命令默认显示连接记录在两个文件中的所有域，而且是按顺序来显示的。-o选项用于改变结果显示的格式

join命令默认比较文件1和文件2的第1域，如果我们需要通过其他域进行连接，就需要使用-1和-2选项，-1用于设置文件1用于连接的域，-2用于设置文件2用于连接的域

文件1和文件2必须已经根据所要进行连接的域排序

cut命令

cut命令用于从标准输入或文本文件中按域或行提取文本

cut [选项] 文件

选项	意义
-c	指定提取的字符数，或字符范围
-f	指定提取的域数，或域范围
-d	改变域分隔符

paste 命令

paste命令用于将文本文件或标准输出中的内容粘贴到新的文件，它可以将来自于不同文件的数据粘贴到一起，形成新的文件

paste [选项] 文件1 文件2

选项	意义
-d	默认域分隔符是空格或Tab键，设置新的域分隔符
-s	将每个文件粘贴成一行
-	从标准输入中读取数据

paste命令的“-”选项比较特殊，当paste命令从标准输入中读取数据时，“-”选项才起作用

paste命令-选项的用法

```
[root@jselab shell-book]# ls | paste -d" " - - - - -              #从标准输入读取数据
anotherres.sh array_eval2.sh colon.sh example execerr.sh        #每行显示5个文件名
execin.sh exec.sh FILE1 FILE2 forever.sh
hfile loggg loggg1 loopalias.sh matrix.sh
newfile nokillme.sh part1 part2 part3
parttotal refor.sh reif.sh selfkill.sh sleep10.sh
sleep55.sh stack.sh subsenv.sh subsep.sh subsig.sh
subsparallel.sh subspipe.sh subsvar.sh TEACHER.db test.sh
testvar.sh traploop.sh
[root@jselab shell-book]#
```

split 命令

split命令用于将大文件切割成小文件，split命令可以按照文件的行数、字节数切割文件，并能在输出的多个小文件中自动加上编号

split [选项] 待切割的大文件 输出的小文件

选项	意义
-或-l	此两个选项等价，都用于指定切割成小文件的行数
-b	指定切割成小文件的字节
-C	与-b选项类似，但是，切割时尽量维持每行的完整性

tr 命令

tr命令实现字符转换功能，其功能类似于sed命令，但是，tr命令比sed命令简单

tr命令能实现的功能，sed命令都可以实现

tr [选项] 字符串1 字符串2 <输入文件

tr命令要么将输入文件重定向到标准输入，要么从管道读入数据,记住tr命令的输入文件之前需要加上“<”符号

选项	意义
-c	选定字符串1中字符集的补集，即反选字符串1中的字符集
-d	删除字符串1中出现的所有字符
-s	删除所有重复出现的字符序列，只保留一个

tar 命令

tar命令是Linux的归档命令，tar命令可以将文件或目录打成一个包

tar [选项] 文件名或目录名

选项	意义
-c	创建新的包
-r	为包添加新的文件
-t	列出包内容
-u	更新包中的文件，若包中无此文件，则将该文件添加到包中
-x	解压缩文件
-f	使用压缩文件或设备，该选项通常是必选的
-v	详细报告tar处理文件的信息
-z	用gzip压缩和解压缩文件，若加上此选项创建压缩包， 那么解压缩时也需要加上此选项

tar命令的另一重要功能就是解压缩，以下两种解压缩命令足以满足一般应用要求：

tar -xvf 压缩包名称 #解压非gzip格式的压缩包

tar -zxvf 压缩包名称 #解压gzip格式的压缩包

gzip命令是Linux系统中常用的压缩工具，它可以对tar命令创建的包进行压缩，但是，gzip所生成的压缩包使用tar -zxvf命令就可解压缩

第六章 变量和引用

变量的定义

变量用于保存有用信息，如路径名、文件名、数字等，Linux用户使用变量定制其工作环境，使系统获知用户相关的配置。变量本质上是存储数据的一个或多个计算机内存地址。

变量可分为三类：

本地变量是仅可以在用户当前Shell生命期的脚本中使用的变量，类似于C、C++、Java等编程语言中局部变量

环境变量则适用于所有由登录进程所产生的子进程，环境变量在用户登录后到注销之前的所有编辑器、脚本、程序和应用中都有效

位置参数也属于变量，它用于向Shell脚本传递参数，是只读的

变量替换和赋值

变量是某个值的名称，引用变量值就称为变量替换

$符号是变量替换符号，如variable是变量名，那么$variable就表示变量的值

变量赋值有两种格式：

} variable=value

} ${ variable=value }

等号的两边可以有空格，这不影响赋值操作；

如果值（value）中包含空格，则必须用双引号括起来；

变量名只能包括大小写字母（a-z和A-Z）、数字（0-9）、下划杠（_）等符号，并且变量名不能以数字开头，否则视为无效变量名

利用unset命令可以清除变量的值，命令格式为：

unset 变量名

readonly可将变量设置为只读，变量一旦设置为只读，任何用户不能对此变量进行重新赋值

variable=value #先对一个变量进行赋值

readonly variable #将variable变量设置为只读

无类型的shell脚本变量

Shell脚本变量是无类型的，这与awk变量是一样的

bash Shell不支持浮点型，只支持整型和字符型，默认情况下，Shell脚本变量是字符型的，同时，字符型的变量还具有一个整型值，为0；但是，bash Shell并不要求在定义一个变量时声明其类型

Shell会根据上下文判断出数值型的变量，并进行变量的算术运算和比较等数值操作。判断标准是变量中是否只是包含数字，如果变量只包含数字，则Shell认定该变量是数值型的，反之，Shell认定该变量是字符串

环境变量

定义环境变量的方法

ENVIRON-VARIABLE=value　　#环境变量赋值

export ENVIRON-VARIABLE　　#声明环境变量

清除环境变量同样是unset命令

env命令可以列出已经定义的环境变量

PWD和OLDPWD

PWD记录当前的目录路径，当利用cd命令改变当前目录时，系统自动更新PWD的值

OLDPWD记录旧的工作目录，即用户所处的前一个目录

PATH

PATH就记录了一系列的目录列表，Shell为每个输入命令搜索PATH中的目录列表

HOME

HOME记录当前用户的根目录

SHELL

SHELL变量保存缺省Shell，缺省的值为/bin/bash

USER和UID

USER和UID是保存用户信息的环境变量，USR表示已登录用户的名字，UID则表示已登录用户的ID

PPID

PPID是创建当前进程的进程号，即当前进程的父进程号

PS1和PS2

提示符变量，用于设置提示符格式

S1是用于设置一级Shell提示符的环境变量，也称为主提示符字符串，即改变： [root @jselab ~]#

PS1变量是[\u@\h \W]\$，\u、\h、\W和\$表示了特定含义，\u表示当前用户名，\h表示表示主机名，\W表示当前目录名，如果是root用户，\$表示#号，其他用户，\$则表示$号

PS2是用于设置二级Shell提示符的环境变量

提示符变量中特殊符号及其意义

模式	意义
\d	以“周 月 日”格式显示的日期
\H	主机名和域名
\h	主机名
\s	Shell的类型名称
\T	以12小时制显示时间，格式为：HH:MM:SS
\t	以24小时制显示时间，格式为：HH:MM:SS
\@	以12小时制显示时间，格式为：am/pm
\u	当前的用户名
\v	bash Shell的版本号
\V	bash Shell的版本号和补丁号
\w	当前工作目录的完整路径
\W	当前工作目录名字
\#	当前命令的序列号
\$	如果UID为0，打印#；否则，打印$

几个重要的配置文件

$HOME/.bash_profile是最重要的配置文件，当某Linux用户登录时，Shell会自动执行.bash_profile文件，如果.bash_profile文件不存在，则自动执行系统默认的配置文件/etc/profile

```
[gridsphere@jselab ~]# cat .bash_profile          #gridsphere用户的.bash_profile文件
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs

export JAVA_HOME=/usr/local/jdk1.5.0_04
export CATALINA_HOME=/home/gridsphere/jakarta-tomcat-5.0.28
export GLOBUS_LOCATION=/usr/local/globus-4.0.4
export CLASSPATH=/usr/local/jdk1.5.0.04/lib
export ANT_HOME=/usr/local/apache-ant-1.7.0
export PATH=$PATH:$JAVA_HOME/bin: $ANT_HOME/bin: $HOME/bin
```

如果要使新加入的行立即生效，需要利用source命令执行.bash_profile文件。source命令也称为“点命令”，即句点符号“.”和source命令是等价的，source命令通常用于重新执行刚修改的初始化文件，使之立即生效，而不必注销并重新登录

```
        . .bash_profile        #注意：句点符号后面用空格与文件名相分隔
        source bash_profile
```

bash Shell的.bash_login文件来源于C Shell的.login文件，bash Shell的.profile文件来源于Bourne Shell和Korn Shell的.profile文件

当用户登录时，首先查找是否存在.bash_profile文件，若它不存在，则查找是否存在.bash_login文件，若它也不存在，则查找是否存在. profile文件

位置参数

位置参数（positional parameters）是一种特殊的Shell变量，用于从命令行向Shell脚本传递参数

$1表示第1个参数、$2表示第2个参数等等，$0脚本的名字，从${10}开始，参数号需要用大括号括起来，如${10}、${11}、${100}……

$*和$@一样，表示从$1开始的全部参数

特殊位置参数	意义
$#	传递到脚本的参数数量
$*和$@	传递到脚本的所有参数
$$	脚本运行的进程号
$?	命令的退出状态，0表示没有错误，非0表示有错误

引用

引用指将字符串用引用符号括起来，以防止特殊字符被Shell脚本重解释为其他意义，特殊字符是指除了字面意思之外还可以解释为其他意思的字符

符号	名称	意义
“”	双引号	引用除美圆符号（$）、反引号（`）和反斜线（\）之外的所有字符
‘’	单引号	引用所有字符
``	反引号	Shell将反引号中内容解释为系统命令
\	反斜线	转义符，屏蔽下一个字符的特殊意义

全部引用和部分引用

双引号引用除美圆符号（$）、反引号（`）和反斜线（\）之外的所有字符，即$、`和\在双引号中仍被解释为特殊意义

在双引号中保持$符号的特殊意义可以引用变量，如“$variable”表示以变量值替换变量名

用双引号引用变量能够防止字符串分割，保留变量中的空格

单引号引用了所有字符，即单引号中字符除单引号本身之外都解释为字面意义，单引号不再具备引用变量的功能

通常将单引号的引用方式称为全引用，将双引号的引用方式称为部分引用

命令替换

命令替换的两种格式：

`Linux 命令`

$(Linux 命令)

转义

反斜线符号（\）表示转义，当反斜线后面的一个字符具有特殊意义时，反斜线将屏蔽下一个字符的特殊意义，而已字面意义解析它

特殊字符	意义
&	传递到脚本的参数数量
*	0个或多个在*字符之前那个普通字符
+	匹配1个或多个在其之前的那个普通字符
^	匹配行首，或后面字符的非
$	命令的退出状态，0表示没有错误，非0表示有错误
`	反引号，Shell引用符号
“	双引号，Shell引用符号
|	管道符号或表示“或”意义
?	匹配0个或1个在其之前的那个普通字符
\	转义符

echo命令的-e选项表示将转义符后跟字符形成的特殊字符解释成特殊意义

符号	意义
\n	新的一行
\r	返回
\t	表示Tab键
\v或\f	换行但光标仍旧停留在原来的位置
\b	退格键（Backspace）
\a	发出警报声
\0xx	ASCII码0xx所对应的字符

第七章 退出、测试、判断及操作符

退出状态

在Linux系统中，每当命令执行完成后，系统都会返回一个退出状态。该退出状态用一整数值表示，用于判断命令运行正确与否。

} 若退出状态值为0，表示命令运行成功

} 若退出状态值不为0时，则表示命令运行失败

} 最后一次执行的命令的退出状态值被保存在内置变量“$?”中，所以可以通过echo语句进行测试命令是否运行成功

测试结构

测试命令可用于测试表达式的条件的真假。如果测试的条件为真，则返回一个 0值；如果测试的条件为假，将返回一个非 0整数值。

测试命令有两种结构：

} 一种命令是使用test命令进行测试，该命令的格式为：

test expression

其中条件expression是一个表达式，该表达式可由数字、字符串、文本和文件属性的比较，同时可加入各种算术、字符串、文本等运算符。

} 另一种命令格式：

[expression]

其中“[”是启动测试的命令，但要求在expression后要有一个 “]”与其配对。使用该命令时要特别注意“[”后和“]”前的空格是必不可少的。

整数运算符

整数比较运算符是算术运算中很简单的一种，用于两个值的比较，测试其比较结果是否符合给定的条件。

例如：

a –eq b

如果满足a等于b，则测试的结果为真（测试条件用0表示）

如果a不等于b，则测试结果为假（测试条件用非0表示）

测试时有两种格式：

（1） test "num1" numeric_operator "num2“

（2） ["num1" numeric _operator "num2"]

整数比较运算符	描述
num1 -eq num2	如果 num1等于num2，测试结果为0
num1 -ge num2	如果 num1大于或等于num2，测试结果为0
num1 -gt num2	如果 num1大于num2，测试结果为0
num1 -le num2	如果 num1小于或等于num2，测试结果为0
num1 -lt num2	如果 num1小于num2，测试结果为0
num1 -ne num2	如果 num1不等于num2，测试结果为0

字符串运算符

字符串运算符用于测试字符串是否为空、两个字符串是否相等或者是否不相等

字符串运算符	描述
string	测试字符串string是否不为空
-n string	测试字符串string是否不为空
-z string	测试字符串string是否为空
string1 = string2	测试字符串string1是否与字符串string2相等
string1 != string2	测试字符串string1是否与字符串string2 不相等

文件操作符

文件运算符	描述
-d file	测试file是否为目录
-e file	测试file是否存在
-f file	测试file是否为普通文件
-r file	测试file是否是进程可读文件
-s file	测试file的长度是否不为0
-w file	测试file是否是进程可写文件
-x file	测试file是否是进程可执行文件
-L file	测试file是否符号化链接

逻辑运算符

逻辑运算符主要包括逻辑非、逻辑与、逻辑或运算符

逻辑操作符	描述
！ expression	如果expression为假，则测试结果为真
expression1 –a expression2	如果expression1和expression同时为真，则测试结果为真
expression1 –o expression2	如果expression1和expression2中有一个为真，则测试条件为真

下表是逻辑运算符的“真假表”，其中expr1和expr2为表达式，用于描述了一个测试条件

expr1	**expr2**	**! expr1**	**! expr2**	**expr1 –a expr2**	**expr1 –o expr2**
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

简单if结构

简单的if结构是：

```
if  expression
then
    command
    command
    …
fi
```

在使用这种简单if结构时，要特别注意测试条件后如果没有“；”，则then语句要换行，否则会产生不必要的错误。如果if和then可以处于同一行，则必须用“;”

If/else结构

命令是双向选择语句，当用户执行脚本时如果不满足if后的表达式也会执行else后的命令，所以有很好的交互性。其结构为：

```
if expression1
then
```

```
        command
        …
        command
    else
        command
        …
        command
    fi
```

if/elif/else结构

if/elif/else结构针对某一事件的多种情况进行处理。通常表现为“如果满足某种条件，则进行某种处理，否则接着判断另一个条件，直到找到满足的条件，然后执行相应的处理”。其语法格式为：

```
if expression1
  then
     command
     command
     …
  elif expression2
  then
     command
    command
     …
elif expressionN
then
  command
  …
  command
else
  command
  …
  command
fi
```

case结构

f/elif/else结构一样，case结构同样可用于多分支选择语句，常用来根据表达式的值选择要执行的语句，该命令的一般格式为：

```
Variable  in
value1)
  command
…
command;;
value2)
  command
  …
  command;;
…
valueN)
```

```
    command
    …
    command;;
*)
    command
    …
    command;;
esac
```

算术运算符

在Linux Shell中，算术运算符包括：+（加运算）、-（减运算）、*（乘运算）、/（除运算）、%（取余运算）、**（幂运算），这些算术运算符的举例及其结果如下表所示：

运算符	举例	结果
+(加运算)	3+5	8
-(减运算)	5-3	2
*(乘运算)	5*3	15
/(除运算)	8/3	2
%(取余运算)	15%4	3
**(幂运算)	5**3	125

位运算符

位运算符在Shell编程中很少使用，通常用于整数间的运算，位运算符是针对整数在内存中存储的二进制数据流中的位进行的操作

运算符	举例	**解释和value值**
<< （左移）	value=4<<2	4左移2位，value值为16
>> （右移）	value=8>>2	8右移2位，value值为2
& （按位与）	value=8&4	8按位与4，value值为0
| （按位或）	value=8|4	8按位或4，value值为12
~ （按位非）	value=~8	按位非8，value值为-9
^ （按位异或）	value=10^3	10按位异或3，value值为9

自增自减运算符

自增自减操作符主要包括前置自增（++variable） 、前置自减（--variable）、后置自增（variable++）和后置自减（variable--）。

} 前置操作首先改变变量的值（++用于给变量加1，--用于给变量减1），然后在将改变的变量值交给表达式使用

} 后置变量则是在表达式使用后再改变变量的值

要特别注意自增自减操作符的操作元只能是变量，不能是常数或表达式，且该变量值必须为整数型，例如：++1、（num+2）++都是不合法的

数字常量

Linux Shell脚本或命令默认将数字以十进制的方式进行处理，如果要使用其他进制的方式进行处理，则需对这个数字进行特定的标记或加前缀。

} 当使用0作为前缀时表示八进制
} 当使用0x进行标记时表示十六进制
} 同时还可使演示用num#这种形式进行标记进制数

第八章 循环与结构化命令

for循环

列表for循环

列表for循环语句用于将一组命令执行已知的次数，下面给出了for循环语句的基本格式：

```
for variable in {list}
do
    command
    command
    …
done
```

其中do和done之间的命令称为循环体，执行次数和list列表中常数或字符串的个数相同。

当执行for循环时，首先将in后的list列表的第一个常数或字符串赋值给循环变量，然后执行循环体；接着将list列表中的第二个常数或字符串赋值给循环变量，再次执行循环体，这个过程将一直持续到list列表中无其他的常数或字符串，然后执行done命令后的命令序列。

不带列表for循环

不带列表的for循环执行时由用户指定参数和参数的个数，下面给出了不带列表的for循环的基本格式：

```
for variable
do
    command
    command
    …
done
```

其中do和done之间的命令称为循环体，Shell会自动的将命令行键入的所有参数依次组织成列表，每次将一个命令行键入的参数显示给用户，直至所有的命令行中的参数都显示给用户。

类C风格的for循环

类C风格的for循环也可被称为计次循环，一般用于循环次数已知的情况，下面给出了类C风格的for循环的语法格式：

```
for(( expr1; expr2; expr3 ))
do
    command
    command
    …
done
```

对类C风格的for循环结构的解释：

} 其中表达式expr1为循环变量赋初值的语句
} 表达式expr2决定是否进行循环的表达式，当判断expr2退出状态为0执行do和done之间的循环体，当退出状态为非0时将退出for循环执行done后的命令
} 表达式expr3用于改变循环变量的语句

} 类C风格的for循环结构中，循环体也是一个块语句，要么是单条命令，要么是多条命令，但必须包裹在do和done之间。

while循环

while循环语句也称前测试循环语句，它的循环重复执行次数，是利用一个条件来控制是否继续重复执行这个语句。while语句与for循环语句相比，无论是语法还是执行的流程，都比较简明易懂。while循环格式如下：

```
while expression
do
    command
    command
    …
done
```

while循环语句之所以命名为前测试循环，是因为它要先判断此循环的条件是否成立，然后才作重复执行的操作。也就是说，while循环语句执行过程是：先判断expression的退出状态，如果退出状态为0，则执行循环体，并且在执行完循环体后，进行下一次循环，否则退出循环执行done后的命令。

为了避免死循环，必须保证在循环体中包含循环出口条件，即存在expression的退出状态为非0的情况。

计数器控制的while循环

假定该种情形是在已经准确知道要输入的数据或字符串的数目，在这种情况下可采用计数器控制的while循环结构来处理。这种情形下while循环的格式如下所示：

```
 counter = 1
 while expression
do
     command
     …
     let command to operate counter
     command
     …
Done
```

结束标记控制的while循环

在Linux Shell编程中很多时候不知道读入数据的个数，但是可以设置一个特殊的数据值来结束while循环，该特殊数据值称为结束标记，其通过提示用户输入特殊字符或数字来操作。当用户输入该标记后结束while循环，执行done后的命令。在该情形下，while循环的形式如下所示：

```
read variable
while [[ "$variable" ！ = sentinel ]]
do
    read variable
done
```

标志控制的while循环

标志控制的while循环使用用户输入的标志的值来控制循环的结束，这样避免了用户不知到循环结束标记的麻烦。在该情形下，while循环的形式如下所示：

```
signal=0
while (( signal != 1 ))
do
        …
        if expression
        then
```

```
                signal=1
            fi
            …
        Done
```

命令行控制的while循环

有时需要使用命令行来指定输出参数和参数个数，这时用其他的三种形式的while循环是无法实现的，所以需要使用命令行控制的while循环。该形式下，while循环通常与shift结合起来使用，其中shift命令使位置变量下移一位（即$2代替$1，$3代替$2），并且使$#变量递减，当最后一个参数也显示给用户后，$#就会等于0，同时$*也等于空，下面是该情形下，while循环的形式为：

```
while [[ "$*" != "" ]]
do
    echo "$1"
    shift
done
```

until循环

在执行while循环时，只要是expression的退出状态为0将一直执行循环体。until命令和while命令类似，但区别是until循环中expression的退出状态不为0将循环体一直执行下去，直到退出状态为0，下面给出了until循环的结构：

```
until expression
do
    command
    command
    …
Done
```

嵌套循环

一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。在外部循环的每次执行过程中都会触发内部循环，直至内部完成一次循环，才接着执行下一次的外部循环。for循环、while循环和until循环可以相互嵌套。

break循环控制符

break语句可以应用在for、while和until循环语句中，用于强行退出循环，也就是忽略循环体中任何其他语句和循环条件的限制。

continue循环控制符

continue循环控制符应用在for、while和until语句中，用于让脚本跳过其后面的语句，执行下一次循环。

select结构

Select是bash的扩展结构，用于交互式菜单显示，用于从一组不同的值中进行选择，功能有点类似于case结构，但其交互性要比case好的多，其基本结构为：

```
select variable in {list}
do
    command
    …
    break
done
```

第九章 变量的高级用法

内部变量

BASH

BASH记录了bash Shell的路径，通常为/bin/bash，内部变量SHELL就是通过BASH的值确定当前Shell的类型

BASH_SUBSHELL

BASH_SUBSHELL记录了子Shell的层次，这个变量在bash版本3之后才出现的，将在12章介绍

BASH_VERSINFO

BASH_VERSINFO是一个数组，包含6个元素，这6个元素用于表示bash的版本信息

BASH_VERSION

Linux系统的bash Shell版本，包含了主次版本、补丁级别、编译版本和发行状态，即BASH_VERSINFO数组从0到4的值

DIRSTACK

Linux目录栈用于存放工作目录，便于程序员手动控制目录的切换，bash Shell定义了两个系统命令pushd和popd

pushd命令用于将某目录压入目录栈，同时将当前工作目录切换到入栈的目录

popd命令将栈顶目录弹出，栈顶元素变为下一个元素，同时将当前工作目录切换到栈弹出的目录

DIRSTACK记录了栈顶目录值，初值为空

GLOBIGNORE

GLOBIGNORE是由冒号分隔的模式列表，表示通配（globbing）时忽略的文件名集合

GROUPS

GROUPS记录了当前用户所属的群组，Linux的一个用户可同时包含在多个组内，因此，GROUPS是一个数组，数组记录了当前用户所属的所有群组号

HOSTNAME

记录了Linux主机的名字

HOSTTYPE和MACHTYPE

记录系统的硬件架构

OSTYPE

记录了操作系统类型，Linux系统中，$OSTYPE=linux

REPLY

REPLY变量与read和select命令有关

read命令用于读取标准输入（stdin）的变量值

read *variable* *#varibale是变量名*

read将读到的标准输入存储到variable变量中。read命令也可以不带任何变量名，此时，read就将读到的标准输入存储到REPLY变量中

bash Shell的select命令源自于Korn Shell，是一种建立菜单的工具，它提供一组字符串供用户选择，用户不必完整地输入字符串，而只需输入相应的序号进行选择

```
select variable in list
do
  Shell命令1
  Shell命令2
  Shell命令3
……
break
done
```

select自动将list形成有编号的菜单，用户输入序号以后，将该序号所对应list中的字符串赋给variable变量，而序号值则保存到REPLY变量中

SECONDS

记录脚本从开始执行到结束所耗费的时间，以秒为单位

SHLVL

记录了bash Shell嵌套的层次，一般来说，我们启动第一个Shell时，$SHLVL=1，如果在这个Shell中执行脚本，脚本中的SHLVL为2，如果脚本再执行子脚本，子脚本中的SHLVL就变为3

TMOUT

设置Shell的过期时间，当TMOUT不为0时，Shell在TMOUT秒后将自动注销。TMOUT放在脚本中，可以规定脚本的执行时间

SHELLOPTS

Shell选项（options）用于设定bash Shell所支持的一些特性，一个Shell选项有“开”和“关”两种状态

set命令用于打开或关闭选项

```
set -o optionname          #打开名为optionname选项
set +o optionname          #关闭名为optionname选项
```

SHELLOPTS记录了处于“开”状态的Shell选项（options）列表，它是一个只读变量

set命令还可以直接利用选项的简写来开启或关闭选项

bash Shell选项、简写及其意义

选项名称	简写	意义
noclobber	C	防止重定向时覆盖文件
allexport	a	export所有已定义的变量
norify	b	后台作业运行结束时，发送通知
errexit	e	当脚本发生第一个错误时，退出脚本
noglob	f	禁止文件名扩展，即禁用通配（globbing）
interactive	i	使脚本以交互模式运行
noexec	n	读取脚本中的命令，进行语法检查，但不执行这些命令
POSIX	o posix	修改bash及其调用脚本的行为，使其符合POSIX标准
privileged	p	以suid身份运行脚本
restricted	r	以受限模式运行脚本
stdin	s	从标准输入（stdin）中读取命令
nounset	u	当使用未定义变量时，输出错误信息，并强制退出
verbose	v	在执行每个命令之前，将每个命令打印到标准输出（stdout）
xtrace	x	与verbose相似，但是打印完整命令
无	D	列出双引号内以$为前缀的字符串，但不执行脚本中的命令
无	c …	从…中读取命令
无	t	第一条命令执行结束就退出
无	-	选项结束标志，后面跟上位置参数（positional parameter）

字符串处理

计算字符串长度

${#string}

expr length $string

索引命令

expr index $string $substring

匹配子串

expr match $string $substring #在string的开头匹配substring字符串

抽取子串

#{string:position} #从名称为$string的字符串的第$position个位置开始抽取子串

#{string:position:length} #从名称为$string的字符串的第$position个位置开始抽取长度为$length的子串

注意：#{...}格式的命令从0开始对名称为$string的字符串进行标号

#{string: -position} #冒号和横杠符号之间有一个空格符

#{string:(position)} #冒号和左括号之间未必要有空格符

expr substr $string $position $length #从名称为$string的字符串的第$position个位置开始抽取长度为$length的子串

注意： expr substr命令是从1开始对名称为$string的字符串进行标号的

删除子串

} ${string#substring} #删除string开头处与substring匹配的最短子串

} ${string##substring} #删除string开头处与substring匹配的最长子串

} ${string%substring} #删除string结尾处与substring匹配的最短子串

} ${string%%substring} #删除string结尾处与substring匹配的最长子串

替换子串

} 替换子串命令都是${...}格式，可以在任意处、开头处和结尾处替换满足条件的子串

} ${string/substring/replacement} #仅替换第一次与substring相匹配的子串

} ${string//substring/replacement} #替换所有与substring相匹配的子串

} ${string/#substring/replacement} #替换string开头处与substring相匹配的子串

} ${string/%substring/replacement} #替换string结尾处与substring相匹配的子串

有类型变量

Shell变量一般是无类型的，但是bash Shell提供了declare和typeset两个命令用于指定变量的类型，两个命令是完全等价的

declare [选项] 变量名

选项名	意义
-r	将变量设置为只读属性
-i	将变量定义为整型数
-a	将变量定义为数组
-f	显示此脚本前定义过的所有函数名及其内容
-F	仅显示此脚本前定义过的所有函数名
-x	将变量声明为环境变量

双小括号方法，即((...))格式，也可以用于算术运算

双小括号方法也可以使bash Shell实现C语言风格的变量操作

declare命令的-x选项将变量声明为环境变量，相当于export命令，但是，declare -x允许在声明变量为环境变量的同时给变量赋值，而export命令不支持此功能。类似指针

declare -x variable-name=value

间接变量引用

如果第一个变量的值是第二个变量的名字，从第一个变量引用第二个变量的值就称为间接变量引用

} variable1=variable2

} variable2=value

variable1的值是variable2，而variable2又是变量名，variable2的值为value，间接变量引用是指通过variable1获得变量值value的行为

bash Shell提供了两种格式实现间接变量引用

} eval tempvar=\$$variable1

} tempvar=${!variable1}

第十章 I/O重定向

管道

管道是Linux编程中最常用的技术之一，Shell编程中管道符号是竖杠符号：“|”

command1 | command2 | command3 | … |commandn

command1到commandn表示Linux的n个命令，这n个命令利用管道进行通信

cat和more命令都用来显示文件内容，cat命令在显示文件时不提供分页功能，而more命令在显示超过一页的文件时提供了分页功能

cat命令还可以同时显示多个文件，命令格式如下：

cat file1 file2 file3 … filen

重定向

I/O重定向是一个过程，这个过程捕捉一个文件、或命令、或程序、或脚本、甚至代码块（code block）的输出，然后把捕捉到的输出，作为输入发送给另外一个文件、或命令、或程序、或脚本

文件标识符是从0开始到9结束的整数，指明了与进程相关的特定数据流的源

Linux系统启动一个进程（该进程可能用于执行Shell命令）时，将自动为该进程打开三个文件：标准输入、标准输出和标准错误输出，分别由文件标识符0、1、2标识

下图描述了stdin、stout、stderr和Shell命令的关系，Shell命令从标准输入读取输入数据，将输出送到标准输出，如果该命令在执行过程中发生错误，则将错误信息输出到标准错误输出

默认情况下，标准输入与键盘输入相关联，标准输出和标准错误输出与显示器相关联

符号	意义
cmd1 | cmd2	管道符，将cmd1的标准输出作为cmd2的标准输入
> filename	将标准输出写到文件filename之中
< filename	将文件filename的内容读入到标准输入之中
>> filename	将标准输出写到文件filename之中，若filename文件已存在，则将标准输出追加到filename已有内容之后
>| filename	即使noclobber选项已开启，仍然强制将标准输出写到文件filename之中，即将filename文件覆盖掉
n>| filename	即使noclobber选项已开启，仍然强制将FD为n的输出写到文件filename之中，即将filename文件覆盖掉
n> filename	将FD为n的输出写到文件filename之中
n< filename	将文件filename的内容读入到FD n之中
n>> filename	将FD为n的输出写到文件filename之中，若filename文件已存在，则将FD为n的输出追加到filename已有内容之后
<<delimiter	此处文档（Here-document）

cat和>符号结合成为简易文本编辑器

} cat命令后不加任何参数时，cat命令的输入是标准输入，即键盘输入

} 利用I/O重定向符号 “>”将键盘输入写入文件

} cat > newfile后，就可输入需要写到newfile的内容，最后按CTRL+D结束对newfile的编辑

>>符号用于在已有文件后追加一些文本

>|符号是强制覆盖文件的符号，它与Shell的noclobber选项有关系，如果noclobber选项开启，表示不允许覆盖任何文件，而>|符号则可以不管noclobber选项的作用，强制将文件覆盖

重定向标准错误输出，需要使用文件标识符2

2> newfile

<是I/O重定向的输入符号，它可将文件内容写到标准输入之中

<<delimiter符号称为此处文档（Here-document），delimiter称为分界符，该符号表明：Shell将分界符delimiter之后直至下一个delimiter之前的所有内容作为输入

exec命令可以通过文件标识符打开或关闭文件，也可将文件重定向到标准输入，及将标准输出重定向到文件

} execin.sh脚本使用exec将stdin重定向到文件

} execout.sh脚本将stdout重定向到文件

符号	意义
n>&m	将FD为m的输出拷贝到FD为n的文件
n<&m	将FD为m的输入拷贝到FD为n的文件
n>&-	关闭FD为n的输出
n<&-	关闭FD为n的输入
&>file	将标准输出和标准错误输出重定向到文件

代码块重定向是指在代码块内将标准输入或标准输出重定向到文件，而在代码块之外还是保留默认状态

代码块重定向是指对标准输入或标准输出的重定向只在代码块内有效

可以重定向的代码块可以是while、until、for等循环结构，也可以是if/then测试结构，甚至可以是函数

代码块输入重定向符号是<，输出重定向符号是>

} rewhile.sh脚本演示while循环的重定向

} refor.sh脚本演示for循环的重定向

代码块重定向在一定程度上增强了Shell脚本处理文本文件的灵活性，它可以让一段代码很方便地处理一个文件（只要将该文件输入重定向到该代码块）。

命令行处理

命令行处理解释了Shell如何处理一个命令的内部机制

Shell从标准输入或脚本读取的每一行称为管道（pipeline），每一行包含一个或多个命令，这些命令用管道符隔开，Shell对每一个读取的管道都按照下面的步骤处理：

1. 将命令分割成令牌（token），令牌之间以元字符分隔，Shell的元字符集合是固定不变的，包括空格、Tab键、换行字符、分号（;）、小括号、输入重定向符（<）、输出重定向符（>）、管道符（|）和&符号，令牌可以是单词（word）、关键字，也可以是I/O重定向器和分号。
2. 检查命令行的第一个令牌是否为不带引号或反斜杠的关键字，如果此令牌是开放关键字，开放关键字指if、while、for或其他控制结构中的开始符号，Shell就认为此命令是复合命令，并为该复合命令进行内部设置，读取下一条命令，再次启动进程。如果此令牌不是复合命令的开始符号，如该令牌是then、else、do、fi、done等符号，这说明该令牌不应该处在命令行的首位，因此，Shell提示语法错误信息。
3. 检查命令行的第一个令牌是否为某命令的别名，这需要将此令牌与别名（alia）列表逐个比较，如果匹配，说明该令牌是别名，则将该令牌替换掉，返回步骤1，否则进入步骤4。这种机制允许别名递归，也允许定义关键字别名，比如可以用下面命令定义while关键字的别名when。

```
alias when=while
```

1. 执行大括号展开，比如h{a,i}t展开为hat或hit。
2. 将单词开头处的波浪号（~）替换成用户的根目录$HOME。
3. 将任何开头为$符号的表达式，执行变量替换。
4. 将反引号内的表达式，执行命令替换。
5. 将$((string))的表达式进行算术运算。
6. 从变量、命令和算术替换的结果中取出命令行，再次进行单词切分，与步骤1不同的是，此时不再用元字符分隔单词，而是使用$IFS分隔单词。
7. 对于*、?、[...]等符号，执行通配符展开，生成文件名。
8. 将第一个单词作为命令，它可以是函数、内建命令和可执行文件。
9. 在完成I/O重定向与其他类似事项后，执行命令。

举例： echo ~/i* $PWD `echo Yahoo Hadop` $((21*20)) > output

1.Shell首先将命令行分割成令牌，分割成的令牌如下，我们在命令行下方用数字标出各个令牌：

echo ~/i* $PWD `echo Yahoo Hadop` $((21*20))

|--1-||--2-| |--3---| |---------4------------| |-----5-----|

需要注意的是，重定向>output虽已被识别，但是它不是令牌，Shell将在后面对I/O重定向进行处理。

2.检查第一个单词echo是否为关键字，显然echo不是开放关键字，所以命令行继续下面的判断。

3.检查第一个单词echo是否为别名，echo不是别名，命令行继续往下处理。

4.扫描命令行是否需要大括号展开，这条命令没有大括号，命令行继续往下处理。

5.扫描命令行是否需要波浪号展开，命令行中存在波浪号，令牌2将被修改，命令行变为如下形式：

echo /root/i* $PWD `echo Yahoo Hadop` $((21*20))

|--1-||----2---| |--3---| |---------4------------| |-----5-----|

6.扫描命令行中是否存在变量，若存在变量，则进行变量替换，该命令行中存在环境变量PWD，因此，令牌3将被修改，命令行变为如下形式：

echo /root/i* /root `echo Yahoo Hadop` $((21*20))

|--1-||----2---| |--3-| |---------4------------| |-----5-----|

7.扫描命令行中是否存在反引号，若存在则进行命令替换，该命令行存在命令替换，因此，令牌4将被修改，命令行变为如下形式：

echo /root/i* /root Yahoo Hadop $((21*20))

|--1-||----2---| |--3-| |-------4-------| |-----5-----|

8.执行命令行中的算术替换，令牌5将被修改，命令行变为如下形式：

echo /root/i* /root Yahoo Hadop 420

|--1-||----2---| |--3-| |-------4-------| |-5-|

9.Shell将对前面所有展开所产生的结果进行再次扫描，依据$IFS变量值对结果进行单词分割，形成如下形式的新命令行：

echo /root/i* /root Yahoo Hadop 420

|--1-||----2---| |--3-| |--4---| |---5--| |-6-|

由于$IFS是空格，因此，命令行被分割为6个令牌，Yahoo Hadop被分成两个令牌。

10.扫描命令行中的通配符，并展开，该命令行中存在通配符*，展开后，命令行变为如下形式：

echo /root/indirect.sh /root/install.log /root/install.log.syslog /root Yahoo Hadop 420

|--1-||-------2--------| |-------3------| |--------4--------------| |--5-| |---6--| |---7--| |-8-|

i*展开为当前目录下所有以i开头的文件，该目录下有三个i开头的文件：indirect.sh、install.log和install.log.syslog，因此，令牌2又被分为令牌2、3和4。

11.此时，Shell已经准备执行命令了，它寻找echo，echo是内建命令。

12.Shell执行echo命令，此时执行>output的I/O重定向，再调用echo命令，显示最后参数。

命令行处理流程图的左侧跳转箭头从执行命令步骤跳转到初始步骤，这正是eval命令的作用

eval命令将其参数作为命令行，让Shell重新执行该命令行，eval的参数再次经过Shell命令行处理的12个步骤

eval在处理简单命令时，与直接执行该命令无甚区别

} 例10-30演示了eval执行复杂命令

} pipe变量赋为管道符， ls $pipe wc -l发生错误：第1步扫描没有发现有管道符，直到第6步变量替换之后命令行才变成ls | wc -l，第9步根据$IFS变量将命令行重新分割成4个令牌，第11步将ls当作命令，后面的3个令牌|、wc和-l被解析为ls命令的参数，由于该目录下没有|和wc等文件或目录，因此，Shell报语法错误

} eval ls $pipe wc –l正确执行，第1轮的结果，ls | wc -l命令行被重新提交到Shell

} 例10-32试图将evalsource文件每行的第1列作为变量名，第2列作为相对应的变量值，evalsource文件的行为：

} var1 APPLE　　最后的效果是：$var1=APPLE

} evalre.sh脚本关键语句eval “${NAME}=${VALUE}”，第1轮结束后命令变为： var1=APPLE；再次将该命令提交到Shell，成功实现var1变量的赋值

} evalre.sh脚本还使用了代码块重定向，实现对evalsource文件的遍历

第十二章 子shell与进程处理

内建命令

父子Shell是相对的，描述了两个Shell进程的fork关系，父Shell指在控制终端或xterm窗口给出提示符的进程，子Shell是由父Shell创建的进程

Shell命令可分为内建命令（built-in command）和外部命令（external command）

} 内建命令是由Shell本身执行的命令

} 外部命令由fork出来的子Shell执行

} 内建命令不创建子Shell，外部命令创建子Shell，因此，内建命令的执行速度要比外部命令快

内建命令就是包含在bash Shell工具包中的命令，内建的英文单词built-in也说明了这一点，内建命令是bash Shell的骨干部分

保留字（reserved words）也是bash Shell的骨干部分，保留字对于bash Shell具有特殊的含义，用来构建Shell语法结构，for、if、then、while、until等都是保留字，但是，保留字本身不是一个命令，而是命令结构的一部分

冒号可以表示永真，相当于TRUE关键字，演示例12-1的colon.sh脚本

冒号可以用于清空一个文件

圆括号结构

圆括号结构能够强制将其中的命令运行在子Shell中，它的基本格式为：

(

command 1

command 2
…
command n
)

圆括号内的n条命令在子Shell中运行，bash版本3之后定义了内部变量BASH_SUBSHELL，该变量记录了子Shell的层次

子Shell只能继承父Shell的一些属性，但是，子Shell不可能反过来改变父Shell的属性

子Shell能够从父Shell继承得来的属性如下：

} 当前工作目录
} 环境变量
} 标准输入、标准输出和标准错误输出
} 所有已打开的文件标识符
} 忽略的信号

子Shell不能从父Shell继承得来的属性，归纳如下：

除了环境变量和.bashrc文件中定义变量之外的Shell变量

未被忽略的信号处理

Shell的限制模式

以前所讲的Shell都是运行在正常模式下的，Shell还有一种模式称为限制模式，简称rsh（Restricted Shell），处于限制模式的Shell下运行一个脚本或脚本片断，将会禁用一些命令或操作

Shell的限制模式是Linux系统基于安全方面的考虑，目的是为了限制脚本用户的权限，并尽可能地减小脚本所带来的危害

bash Shell的限制模式借鉴了Korn Shell的限制性命令和操作，限制的命令和操作包含如下几方面：

} 用cd命令更改当前工作目录的命令
} 更改重要环境变量的值，包括$PATH、$SHELL、$BASH_ENV、$ENV和$SHELLOPTS
} 输出重定向符号，包括>、>>、>|、>&、<>和&>符号
} 调用含有一个或多个斜杠（/）的命令名称
} 使用内建命令exec
} 使用set +r等命令关闭限制模式

使脚本运行在限制性模式下的两种方式：

} set -r命令开启restricted选项，演示resshell.sh脚本
} #!后的语句改成/bin/bash -r，演示anotherres.sh脚本

进程和作业

Unix是第一个允许每个系统用户控制多个进程的操作系统，这种机制称为用户控制的多任务，Linux操作系统延续了此特性

} 一个正在执行的进程称为作业，一个作业可以包含多个进程
} 用户提交作业到操作系统，作业的完成可能依赖于启动多个进程
} 作业是用户层面的概念，而进程是操作系统层面的概念

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，进程在运行中不断地改变其运行状态

} 就绪（Ready）状态：当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态
} 运行（Running）状态：当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态
} 阻塞（Blocked）状态：正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待I/O完成、申请缓冲区不能满足、等待信号等

进程号和作业号

} Linux系统为每个进程分配一个数字以标识这个进程，这个数字就是进程号

} 创建该进程的Shell为此进程创建一个数字，也用于标识这个进程，这个数字称为作业号

} 作业号标识的是在此Shell下运行的所有进程，我们知道，Linux是多用户的系统，多用户可能开启了多个Shell，进程号就标识了整个系统下正在运行的所有进程

} 举例说明：

```
[root@jselab shell-book]# grep -r "root" /etc/* | sort > part1 &
[1] 4693
[root@jselab shell-book]# grep -r "root" /usr/local/* | sort > part2 &
[2] 4695
```

作业控制

作业有两种运行方式：前台运行和后台运行

} 前台运行的作业指作业能够控制当前终端或窗口，且能接收用户的输入

} 后台运行的作业则不在当前激活的终端或窗口中运行，是在用户“看不见”的情况下运行

} 内建命令fg可将后台运行的作业放到前台，而&符号使得作业后台运行

} fg命令通过%n通过作业号指定作业

} 内建命令jobs显示所有后台作业

参数	意义
%n	n为后台作业的作业号
%string	命令以string字符串开始的后台作业
%?string	命令包含string字符串的后台作业
%+或%%	最近提交的后台作业
%-	最近第二个提交的后台作业

CTRL+Z组合键可以使作业转入阻塞态

bg命令可使阻塞态作业转入后台运行

fg、bg和jobs命令只能以作业号为参数来指定作业，这三个命令是不能使用进程号的

disown命令用于从Shell的作业表中删除作业，作业表指得就是由jobs命令所列出的作业列表

wait命令用于等待后台作业完成，subsparallel.sh脚本已经使用到了wait命令

信号

信号是Linux进程间通信的一个重要而复杂的概念，信号是在软件层次上对中断机制的一种模拟

信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达

信号来源：硬件来源和软件来源

组合键	信号类型	意义
CTRL+C	INT信号，即interrupt信号	停止当前运行的作业
CTRL+Z	TSTP信号，即terminal stop信号	使当前运行的作业暂时停止（转入阻塞态）
CTRL+\	QUIT信号	CTRL+C的强化版本，当CTRL+C无法停止作业时，使用此组合键
CTRL+Y	TSTP信号，即terminal stop信号	当进程从终端读取输入数据时，暂时停止该进程

} CTRL+C，INT信号，退出作业，演示例12-19

} CTRL+\,QUIT信号， CTRL+C的强化版本，例12-20

} CTRL+Y组合键实际上与CTRL+Z组合键是类似的，都是向进程发送TSTP信号，表示将进程暂时停止

} kill命令可以通过进程号、作业号或进程命令名向任何作业发送信号

} 演示例12-22，selfkill.sh脚本

} kill –l列出kill命令能发出的所有信号

trap命令

trap是Linux的内建命令，它用于捕捉信号，trap命令可以指定收到某种信号时所执行的命令

} trap command sig1 sig2 …sigN

} trap命令表示当收到sig1、sig2、…、sigN中任意一个信号时，执行command命令，command命令完成后脚本继续收到信号前的操作，直到脚本执行结束

} 演示例12-22， traploop.sh脚本

停此进程：

trap命令还可以忽略某些信号，即进程收到某些信号后不做任何处理，我们只要简单将trap命令的command用空字符串代替即可（""或''）

子Shell继承父Shell忽略的信号

子Shell不能继承父Shell未忽略的信号

第十三章 函数

函数的定义和基本知识

函数是一串命令的集合，函数可以把大的命令集合分解成若干较小的任务。编程人员可以基于函数进一步的构造更复杂的Shell 程序，而不需要重复编写相同的代码。下面给出了Linux Shell中函数的基本形式。

```
function_name()
{
    command1
    command2
    …
    commandN
}
```

对函数结构进行解释：

} 其中标题为函数名，函数体是函数内的命令集合，在编写脚本时要注意标题名应该唯一，如果不唯一，脚本执行时会产生错误。

} 在函数名前可以加上关键字function，但加上和省略关键字function对脚本的最终执行不产生任何影响。

} 函数体中的命令集合必须含有至少一条命令，即函数不允许空命令，这一点和C语言不同。

} 函数之间可通过参数、函数返回值交互，函数在脚本中出现的次序可以是任意的，但是必须按照脚本中的调用次序执行这些函数。

向函数传递参数

在bash Shell编程中，向函数传递的参数仍然是以位置参数的方式来传递的，而不能传递数组等其它形式变量，这与C语言或Java语言中的函数传递是不同的。

下面演示例13-4说明参数传递情况：

} 在Linux Shell编程中，函数还可传递间接参数，但该方式传递方式远远没有C语言和Java语言灵活，而只能使用第9章所述的间接变量引用来传递参数，这种方式是一种笨拙的间接参数传递方式。其中表达式expr1为循环变量赋初值的语句

} 演示例13-6说明间接参数传递方法

函数返回值

有时需要脚本执行完成后返回特定的值来完成脚本的后继操作，这些特定的值就是函数返回值。在Linux Shell编程中，函数通过return返回其退出状态，0表示无错误，1表示有错误。在脚本中可以有选择的使用return语句，因为函数在执行完最后一条语句后将执行调用该函数的地方执行后继操作。

演示例13-7说明函数返回值的用法

脚本放置多个函数

可以在脚本中放置多个函数，脚本执行时按照调用函数的顺序执行这些函数.

演示例13-8讲解多函数的调用顺序

数相互调用

在Linux Shell编程中，函数之间可以相互调用，调用时会停止当前运行的函数转去运行被调用的函数，直到调用的函数运行完，再返回当前函数继续运行。

演示例13-9讲解如何实现函数间的相互调用

一个函数调用多个函数

在Linux Shell编程中允许一个函数调用多个函数，在该函数调用其他函数时同样需要按照调用的顺序来执行调用的函数。

演示13-10讲解如何实现一个函数调用多个函数

局部变量和全局变量

在Linux Shell编程中，可以通过local关键字在Shell函数中声明局部变量，局部变量将局限在函数范围内。此外，函数也可调用函数外的全局变量，如果一个局部变量和一个全局变量名字相同，则在函数中局部变量将会覆盖掉全局变量

演示例13-11讲解局部变量和全局变量的用法

函数递归

Linux Shell中可以递归调用函数，即函数可以直接或间接调用其自身。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。

例13-12是一个递归调用的例子。

```
#!/bin/bash
```

```
#递归调用函数
foo()
{
    read y
    foo $y
    echo "$y"
}

#调用函数
foo
exit 0
```

使用局部变量的递归

使用局部变量进行递归一般是针对数值运行来使用的。阶乘运算是一个使用局部变量的递归调用过程，实现了n! 的运算，其可以通过下面的公式表示：

n!=1　　(n=0)

n!=n*(n-1)!　　(n>=1)

按照该公式可实现对阶乘的运算，由于该阶乘运算中存在终止条件“0!=1”，所以可以使用函数递归来实现该运算。

演示例13-13实现n!的运算

为了观察递归调用的工作过程，下面跟踪下面语句的执行num=3，下面是递归的执行过程：

} num=3：发现num的值不等于0，所以调用函数 fact 3

} num=2：发现num的值不等于0，所以调用函数fact 2

} num=1：发现num的值不等于0，所以调用函数fact 1

} num=0，这是num是等于0的，所以返回调用fact 0，返回factorial的值为1

} num=1，返回factorial的值为1*1=1

} num=2，返回factorial的值为1*2=2

} num=3，返回factorial的值为2*3=6

} 在最终传递到0时，fact函数开始将先前的调用逐个分解，直到num=3的原始调用为止，并返回最终结果为6。

不使用局部变量的递归

使用局部变量的递归一般可通过递推法实现，如上面的阶乘问题可通过1乘以2，再乘以3直到乘以n来得到最终结果，但有些问题只能通过递归来实现，这类问题一般涉及到不使用局部变量的递归，最著名的是汉诺塔问题。

一块板上有三根针A、B和C，A针上套有n个大小不等的圆盘，大的在下，小的在上。要把这n个圆盘从A针移动C针上，每次只能移动一个圆盘，移动可以借助B针进行。但在任何时候，任何针上的圆盘都必须保持大盘在下，小盘在上，求移动的步骤。

设A上有n个盘子，该问题可分解为下面的问题加以解决：

} 如果n=1，则将圆盘从A直接移动到C。

} 如果n=2，则：

1.将A上的n-1(等于1)个圆盘移到B上；

2.再将A上的一个圆盘移到C上；

3.最后将B上的n-1(等于1)个圆盘移到C。

} 如果n=3，则：

1. 将A上的n-1(等于2，令其为n')个圆盘移到B(借助于C)，步骤如下

(1)将A上的n'-1(等于1)个圆盘移到C上。

(2)将A上的一个圆盘移到B。

(3)将C上的n'-1(等于1)个圆盘移到B。

2. 将A上的一个圆盘移到C。

3. 将B上的n-1(等于2，令其为n')个圆盘移到C(借助A)，步骤如下：

(1)将B上的n'-1(等于1)个圆盘移到A。

(2)将B上的一个盘子移到C。

(3)将A上的n'-1(等于1)个圆盘移到C。

从上面分析可以看出，当n大于等于2时，移动的过程可分解为三个步骤：

} 第一步：把A上的n-1个圆盘移到B上；

} 第二步：把A上的一个圆盘移到C上；

} 第三步：把B上的n-1个圆盘移到C上；

其中第一步第三步是类同的，显然这是一个递归过程，所以可以通过Linux Shell编程实现，演示例13-14实现汉诺塔问题。

第十四章 别名、列表及数组

别名

bash Shell的别名（aliases）可以为系统命令重新起一个名字

命令格式是：alias alias-name= 'original-command'

} alias是指定别名命令的关键字

} alias-name是用户所指定的别名（新名字）

} original-command是所起别名所对应命令及其参数，当original-command是以空格分隔的字符串时，就要将original-command用引号括起来

} 等号两边是不能有空格的

删除已经设置的别名，可以使用unalias命令，unalias是一个内建命令，有如下两种格式：

} unalias alias-name　　#删除别名alias-name

} unalias -a　　#删除所有别名

在脚本中使用别名需要打开expand_aliases选项

} shopt -s expand_aliases

} 演示alias.sh脚本

alias命令不能在诸如if/then结构、循环和函数等混合型结构中使用

} 演示loopalias.sh脚本

列表

列表由一串命令用与运算（&&）和或运算（||）连接而成，用与运算连接的列表称为与列表（and list），用或运算连接的列表称为或列表（or list）

命令1 && 命令2 && 命令3 && … && 命令n

} 上述格式的与列表从左至右依次执行命令，直到某命令返回FALSE时，与列表执行终止

} 演示andlist1.sh脚本

命令1 || 命令2 || 命令3 || … || 命令n

} 或列表依然是从左至右依次执行命令，但是，它是当某命令返回TRUE时，或列表执行终止

数组的基本用法

数组（Array）是一个由若干同类型变量组成的集合，引用这些变量时可用同一名字。数组均由连续的存储单元组成，最低地址对应于数组的第一个元素，最高地址对应于最后一个元素

bash Shell只支持一维数组，数组从0开始标号，以array[x]表示数组元素，那么，array[0]就表示array数组的第1个元素、array[1]表示array数组的第2个元素、array[x]表示array数组的第x+1个元素

bash Shell取得数组值（即引用一个数组元素）的命令格式是：

} ${array[x]}　　#引用array数组标号为x的值
} $符号后面的大括号必不可少
} 演示array_eval1.sh脚本
　　还可以用小括号将一组值赋给数组
} 演示array_eval2.sh脚本
} 演示array_eval3.sh脚本
} 演示array_eval4.sh脚本
　　array[@]和array[*]都表示了array数组的所有元素

```
for i in ${city[@]}　　#将@替换为*亦可
do
    echo $i
done
```

　　演示array_print1.sh脚本和array_print2.sh脚本

数组的特殊用法

　　数组的字符串操作
} 操作符号及其意义与9.2节所介绍的字符串操作完全一致，数组字符串操作的特殊之处在于所有操作都是针对所有数组元素逐个进行的
} 演示string_array.sh脚本
　　用read命令从stdin读取一系列的值
} read -a array可以将读到的值存储到array数组
} 演示arrivedcity.sh脚本
　　数组连接
} 将多个数组合并到一个数组
} 演示combine_array.sh脚本

用数组实现简单的数据结构

　　数据结构是指相互之间存在一种或多种特定关系的数据元素的集合，数据结构直接影响到程序的运行速度和存储效率
　　bash Shell不直接支持如堆栈、队列、链表等数据结构，但是，通过数组bash Shell可以很容易地实现线性数据结构
　　讲解用Shell数组实现堆栈结构
} 堆栈的访问规则被限制为Push和Pop两种操作，Push（入栈或压栈）向栈顶添加元素，Pop（出栈或弹出）则取出当前栈顶的元素，也就是说，只能访问栈顶元素而不能访问栈中其它元素
} 如果所有元素的类型相同，堆栈的存储也可以用数组来实现，访问操作可以通过函数接口提供

```
#stack.sh脚本：利用数组实现堆栈
#!/bin/bash

MAXTOP=50                    #堆栈所能存放元素的最大值
TOP=$MAXTOP    #定义栈顶指针，初始值是$MAXTOP
TEMP=
#定义一个临时全局变量，存放出栈元素，初始值为空

declare -a STACK              #定义全局数组STACK
#push函数是进栈操作，可以同时将多个元素压入堆栈
push()
{
if [ -z "$1" ]                #若无任何参数输入，立即返回
```

```
then
  return
fi

#下面的until循环将push函数的所有参数都压入堆栈
until [ $# -eq 0 ]
do
let TOP=TOP-1                    #栈顶指针减1
STACK[$TOP]=$1                   #将第1个参数压入堆栈
shift                            #脚本参数左移1位，$#减1
done

return
}
```

shift命令完成两个功能：

1. 所有位置参数左移1位，即$2移到$1的位置，$3移到$2的位置，依次类推

（2）$#变量值减1。依赖上述的shift命令的两个功能，下一次的until循环就可以对下一个位置参数进行处理，被处理的位置参数标号依然是$1

直到$#变量等于0时，所有位置参数处理完毕，until循环结束，push函数随之结束

```
#pop函数是出栈操作，执行pop函数使得栈顶元素出栈
pop()
{
TEMP=                            #清空临时变量

if [ "$TOP" -eq "$MAXTOP" ]          #若堆栈为空，立即返回
then
 return
fi

TEMP=${STACK[$TOP]}              #栈顶元素出栈
unset STACK[$TOP]
let TOP=TOP+1                    #栈顶指针加1
return
}
```

第十五章 一些混杂的主题

脚本编写风格

如果你编写的脚本超过百行，或者你希望在过一段时间之后，还能够正确理解自己的脚本内容的话，就必须养成良好的脚本编程习惯。在诸多编程习惯当中，编程风格是最重要的一项内容。

良好的编程风格可以在许多方面帮助开发人员。良好的编程风格可以增加代码的可读性，并帮助你理清头绪。如果程序非常杂乱，大概看一下你就会晕头转向。写脚本也是一门艺术，良好清晰的版式，能让人一目了然，让阅读者有清晰的逻辑，Shell编写风格最能体现一个Linux Shell程序员的综合素质。

需要从缩进、{}的格式、空格和空行的用法、判断和循环的编程风格、命名规范、注释风格等角度注意自己的编程风格

缩进

} 缩进格式是4个字符。

} 程序的缩进超过3级，则应考虑重新设计程序

{}的格式

} 一种是“{”括号与“}”括号每个占用一行，缩进与函数名一致，而函数体则缩进4个字符。

} 另外一种格式是：“{”括号与函数名在同一行中，函数体缩进4个字符，而“}”括号则单独占用一行。

空格和空行的用法

} 程序对于空格来说，缩进是空出四个空格

} 对于运算符（赋值运算符除外）来说一般前或空出一个空格。

} 对于赋值运算符“=”左右是不能加空格，否则会发生不必要的错误

} 对于判断运算符来说，一般要加上if后要加空格与“[”运算符隔开

} 对与空行来说，空行起着分割代码的作用，添加必要的空行有时是必须的，一般说来在函数的开始和结束、判断或循环的始末、函数调用始末以及前后联系不紧密的地方都要加空格

判断和循环的编程风格（第七章和第八章已经详细描述了循环和判断的格式这里就不再一一描述)

命名规范

} 对于函数来说，一般是根据函数的功能来进行命名，其一般有两种命名方式：

(1)对于一些复杂点的函数操作，可以使用“操作对象+操作”的形式进行命名，如：array_sort()函数，从字面上就可以看出该函数用于对数字进行排序。

(2)对于简单的函数，可以直接用操作名来做函数名，但要注意不要和系统的命令相同造出不必要的错误，如address()，该函数对进行操作。

} 对于变量来说，一般是通过匈牙利命名法进行命名：

(1)对于单个英文单词就可命名的，可直接用该单词进行命名，如变量average可用于对变量“平均数”进行命名。

(2)对于单个单词无法命名的单词，可通过双单词或多单词形式的缩略词来进行命名，如： dir_num可以用于命名变量“目录个数”。

} 常量来说，可通过将将该变量全部设置为大写与变量形成区别，下面是常量的命名方式：

(1)对于单个单词可以命名的，可直接使用该单词的全部大写形式进行命名，如：常量TOTAL可对常量“总数”进行命名。

(2)对于单个单词无法表达清晰的常数，可通过加下划线的形式对其进行命名，如：常数GLOBAL_CON可对常量“全局常量”进行命名。

脚本优化

遇到重复编写的的代码可创建一个函数来处理这些重复部分，然后通过函数调来简化脚本。

不要对静态值进行硬编码，保持脚本灵活性。演示例15-8。

在编写有参数输入的脚本时，要特别注意给用户足够的提示，提示用户需要输入的参数是什么，同时提供参数个数和参数类型判断，否则用户不了解脚本中设置的参数信息，就可能无法完成该脚本的执行。

Linux中的特殊命令

获取和处理命令行选项的getopts语句，该语句可以编写脚本，使控制多个命令行参数更加容易。该语句的格式为：shift命令主要用于向脚本传递参数使的每一位参数偏移，其每次向将参数位置向左偏移一位。

演示例15-14说明shift命令的用法

Linux Shell中提供了一条

getopts option_str variable

} 在该命令行中， option_str中包含一个有效的单字符选项。

} 若getopts命令在命令行中发现了连字符，那么该命令将用连字符后面的字符同option_str相比较。若匹配成功，则把变量variable的值设为该选项；

} 若匹配不成功，则variable设为“?”。当getopts发现连字符后面没有字符，会返回一个非零的状态值。

} 演示例15-15，可以看出getopts对命令行所给出的选项的分析过程为：

（1）getopts选项检查所有命令行参数，找到以“-”字符开头的字符。

（2）当找到以“-”字符开头的参数后，将跟在“-”字符后的字符与在“option-str”中给出的字符进行比较。

（3）若找到匹配，指定的变量variable被设置成选项，否则，variable被设置成“?”字符。

（4）重复（1）~（3）直到考虑完所有的选项。

（5)当分析结束后，getopts返回非零值并退出。

交互式和非交互式shell脚本

Linux Shell中许多管理和系统维护脚本是非交互式的，而且非多变的重复性任务可以由非交互式脚本完成，由非交互式Shell脚本完成的脚本通常是运行脚本中的命令，而不需要用户干预脚本的执行结果和执行方式。

演示例15-17说明非交换式Shell脚本如何编写

Shell是一个交互性命令解释器。Shell独立于操作系统，这种设计让用户可以灵活选择适合自己的Shell。Shell让你在命令行键入命令，经过Shell解释后传送给操作系统（内核）执行。同时Shell脚本同样也可以通过交互来执行。 。

演示例15-19说明交互式Shell 脚本如何编写

/dev文件系统

Linux中的设备有两种类型，字符设备（无缓冲且只能顺序存取）和块设备（有缓冲且能随机存取）。每个字符设备和块设备都有主、次设备号，主设备号相同的设备是同类设备（使用同一个驱动程序）。这些设备中，有些设备是对实际存在的物理硬件的抽象，而有些设备则是内核自身提供的功能，每个设备在/dev目录下都有一个对应的文件（节点）。

/dev/zero是一个非常有用的伪设备，其可用于创建空文件而且可以创建RAM文件等。

演示例15-21说明/dev/zero的作用

dev/null相当于一个文件的“黑洞”， 它非常接近于一个只写文件，所以写入它的内容都会永远丢失但是, 对于命令行和脚本来说，/dev/null却非常的有用。如果不想使某文件使用stdout，可以通过使用/dev/null将stdout禁止掉，下面的命令就是实现该功能。

```
[root@localhost chapter15]# touch file1.sh
[root@localhost chapter15]# cat file1.sh >/dev/null
```

演示例15-23，说明/dev/null如何隐藏文件信息

/proc文件系统

/proc文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过/proc得到系统的信息，并可以改变内核的某些参数。

演示例15-24，查看中断信息

使用/proc/sys优化系统参数：可通过/proc/sys目录修改内核参数来优化系统，演示例15-25改变文件大小的上限。

查看运行中的进程信息：演示例15-26查找一个正在运算的mozilla进程

查看文件系统信息：演示例15-27查看文件系统支持的类型

查看网络信息：演示15-30获得网络套接字的使用统计

shell包装

Shell包装的脚本指的是内嵌系统命令或工具的脚本，并且这种脚本保留了传递给命令的一系列参数。因为包装脚本中包含了许多带有参数的命令, 使它能够完成特定的目的, 所以这样就大大简化了命令行的输入，尤其适用于sed和awk命令。

演示例15-32，该例包装了sed命令,完成对输入文件中的字符串替换的功能

演示例15-33，用于输出部分ASCII码。

带颜色的脚本

在Linux Shell脚本，脚本执行终端的颜色可以使用“ANSI非常规字符序列”来生成，例如：

```
echo -e "\033[44;37;5m Hello \033[0m World"
```

以上命令设置前景色成为白色，背景色为白色，闪烁光标，输出字符ME，然后重新设置屏幕到缺省设置，输出字符COOL。e是命令echo的一个可选项，它用于激活特殊字符的解析器。\033引导非常规字符序列。m意味着设置属性然后结束非常规字符序列，这个例子里真正有效的字符是“44;37;5”和“0”。修改“44;37;5”可以生成不同颜色的组合，数值和编码的前后顺序没有关系。

演示例15-35，通过不同的颜色体现错误、警告、完成和普通信息的区别

Linux脚本安全

尽管Linux的安全性比Windows系统要好，但并不代码Linux是绝对安全的，所以在Linux Shell编程过程中要注意以下一些问题：

不要将当前目录置于PATH下，可执行脚本应该放在标准的系统目录下，否则将会打开特洛伊木马的大门。

确认PATH下的每个目录都有其对应的拥有者可以写权限，其他任何人不能写入，否则将有可能被病毒侵入的危险。

写程序时要花费时间想想如何去写，在开始运行前要不断的设法测试，在设计时最好将如何设计实现也写入其中。

在编写脚本时最好不要使用root用户，否则有可能被别人窃取密码。

不要在用户输入上使用eval，如果其他用户在读取脚本时发现使用了eval，则可以轻松的将这个脚本破坏掉。

小心的检测自己编写的脚本，寻找是否存在有可能被利用的漏洞和错误，试着找出破坏它的方式，再修正这些发现的问题。

使用shc工具加密shell脚本

如果你的Shell脚本包含了敏感的口令或者其它重要信息，而且不希望用户通过命令ps -ef（查看系统每个进程的状态）捕获敏感信息。 你可以使用shc工具来给shell脚本增加一层额外的安全保护。 shc是一个脚本编译工具，使用RC4加密算法, 它能够把shell程序转换成二进制可执行文件（支持静态链接和动态链接）。该工具能够很好的支持: 需要加密、解密或者通过命令参数传递口令的环境。

在使用shc工具之前，首先需要下载安装包shc-3.8.6.tgz，本书附带光盘也提供了此安装包，演示安装步骤。

演示例15-36对文件进行加密。

Linux shell脚本编写的病毒

可以使用Linux Shell脚本编写病毒。

看一下例15-37中的一个最简单的病毒

看一下例15-38，该例是对15-37中的病毒脚本进行改进

例15-38中的virus2.sh改进了一下，加了若干的判断，判断文件是否存在、是否文件可执行、是否有写权限，再判断它是否是脚本程序如果是就使用cp命令，所以这段代码是能够破坏掉该系统中所有的脚本程序，危害性还是比较大的。

Linux shell中的木马

在Linux Shell中同样存在木马，如特洛伊木马，其看上去是无害的，但其却隐藏着危险的东西，如下面的例子中，将Linux Shell编写的一些脚本放入到~/wyq/bin里，则该目录会出现～/wyq/.profile里path变量的第一个，如果我们将bin目录保留给其他用户使用。可通过下面的脚本内容窃取bin目录下的这些脚本。

```
#nasty_shell.sh：放置在~/wyq/bin下的木马
#!/bin/bash

 /bin/grep "$@"

case $(whoami) in
root)
  #用于窃取操作，这里省略
  …
```

```
#隐藏操作痕迹
rm ~/wyq/bin/ nasty_shell.sh
```

第十六章 shell脚本调试技术

shell脚本调试概述

Shell脚本调试就是发现引发脚本错误的原因以及在脚本源代码中定位发生错误的行，常用的手段包括分析输出的错误信息，通过在脚本中加入调试语句，输出调试信息来辅助诊断错误，利用调试工具等

Shell解释器缺乏相应的调试机制和调试工具的支持，其输出的错误信息又往往很不明确，因此，Shell脚本调试是一个令程序员头痛的问题

Shell脚本的错误可分为两类：

} 语法错误（syntax error），脚本无法执行到底

```
#例16-1：misskey.sh脚本演示漏写关键字错误
#!/bin/bash

var=0
while :
 if [ $var -gt 3]
then
      break
fi
let "var=var+1"
done
```

} Shell脚本能够执行完毕，但是并不是按照我们所期望的方式运行，即存在逻辑错误

```
#例16-2：runsec.sh脚本演示逻辑错误
#!/bin/bash
count=1                    #用于记录进入while循环的次数
MAX=5

while [ "$SECONDS" -le "$MAX" ]
do
echo "This is the $count time to sleep."
        count=$count+1
 sleep 2
done

echo "The running time of this script is $SECONDS"
```

使用trap命令

trap是Linux的内建命令，它用于捕捉信号，trap命令可以指定收到某种信号时所执行的命令

} trap command sig1 sig2 …sigN

Shell脚本在执行时，会产生三个所谓的“伪信号”，（之所以称之为“伪信号”是因为这三个信号是由Shell产生的，而其它的信号是由操作系统产生的）

利用trap命令捕获这三个“伪信号”并输出相关信息是Shell脚本调试的一种重要技巧

信号名称	产生条件
EXIT	从函数中退出，或整个脚本执行完毕
ERR	当一条命令返回非零状态码，即命令执行不成功
DEBUG	脚本中的每一条命令执行之前

在调试过程中，为了跟踪某些变量的值，我们常常需要在Shell脚本的许多地方插入相同的echo语句来打印相关变量的值，这种做法显得烦琐而笨拙。而利用trap命令捕获DEBUG信号，我们只需要一条trap语句就可以完成对相关变量的全程跟踪

} 演示例16-4的trapdebug.sh脚本

从函数退出或脚本结束时，Shell发出EXIT信号；当函数或脚本返回非0值时，Shell发出ERR信号

} 演示例16-5的trapexit.sh脚本

} 演示例16-6的traperr.sh脚本

使用tee命令

tee命令产生的数据流向很像英文字母T，将一个输出分为两个支流，一个到标准输出，另一个到某输出文件

tee命令的这种特性可以用到Shell脚本的管道及输入输出重定向的调试上，当我们发现由管道连接起来的一系列命令的执行结果并非如预期的那样，就需要逐步检查各条命令的执行结果来定位错误，但因为使用了管道，这些中间结果并不会显示在屏幕上，这给调试带来了困难

obtainIP.sh目标是获得机器的IP地址并存储到某变量之中

```
localIP=`cat /etc/sysconfig/network-scripts/ifcfg-eth0 | grep 'IPADDR' | cut -d= -f2`
echo "The local IP is: $localIP“
```

obtainIP.sh目标简洁，但是，不容易发现其中错误，有必要加入tee命令，看清楚中间结果

```
localIP=`cat /etc/sysconfig/network-scripts/ifcfg-eth0 | tee debug.txt | grep 'IPADDR' | tee -a debug.txt | cut -d= -f2 | tee -a debug.txt`
echo "The local IP is: $localIP"
```

调试钩子

调试钩子也称为调试块，是源自于高级程序设计语言中的方法。调试钩子实际上是一个if/then结构的代码块，DEBUG变量控制该代码块是否执行，在程序的开发调试阶段，将DEBUG变量设置为TRUE，使其输出调试信息，到了程序交付使用阶段，将DEBUG设置为FALSE，关闭调试钩子，而无需一一删除调试钩子的代码

```
if [ "$DEBUG" = "true" ]
then
  echo "Debugging information:"
…                    #在此可添加其他输出调试信息
fi
```

演示例16-8的debugblock.sh脚本

使用shell选项

在众多的Shell选项中，有三个选项可以用于脚本的调试，它们是-n、-x和-c

两种方法用-n选项进行语法检查

} set -n

} sh -n 脚本名

} 演示例16-9

选项名称	简写	意义
noexec	n	读取脚本中的命令，进行语法检查，但不执行这些命令
xtrace	x	在执行每个命令之前，将每个命令打印到标准输出（stdout）
无	c …	从…中读取命令

-x选项使Shell在执行脚本的过程中把它实际执行的每一个命令行显示出来，并且在行首显示一个“+”符号，“+”符号后面显示的是经过了变量替换之后的命令行内容，有助于分析实际执行的命令

-x选项经常与trap捕捉DEBUG信号结合使用，这样既可以输出实际执行的每一条命令，又可以逐行跟踪相关变量的值，对调试相当有帮助

} PS4可以改变-x选项提示符，还经常使用两个变量

} FUNCNAME数组变量，表示整个调用链上所有的函数名

} LINENO表示Shell脚本的行号

} 演示例16-12的nestfun.sh脚本

} -c选项使Shell解释器从一个字符串中读取命令，演示例16-13

第十七章 bash编程范例

将文本文件转化为HTML文件

HTML（HyperText Mark-up Language）称为超文本标记语言或超文本链接标示语言，是目前万维网（WWW, World Wide Web）上应用最为广泛的语言，也是构成网页文档的主要语言

HTML文本是由HTML命令组成的描述性文本，HTML命令可以说明文字、图形、动画、声音、表格、链接等。HTML的结构包括头部（Head）、主体（Body）两大部分，其中头部描述浏览器所需的信息，而主体则包含所要说明的具体内容

TEACHER.db文件的每行是一条记录，记录了一位教授的姓名、所在学校、城市和国家，域之间用冒号分隔，我们现在要将TEACHER.db文件转换为HTML格式的文件

```
#TEACHER.html文件的格式
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
 <HEAD>
  <TITLE>
   ……
  </TITLE>
 </HEAD>
 <BODY>
 <TABLE>
<TR>
<TD>B Liu</TD>
<TD>Shanghai Jiaotong University</TD>
<TD>Shanghai</TD>
<TD>China</TD>
</TR>
……
</TABLE>
 </BODY>
 </HTML>
```

<TABLE></TABLE>标签对定义了一张表格，表格由行组成，每行对应于TEACHER.db文件的一条记录

<TR></TR>标签对定义了表格的一行，表格行又包含了很多单元格

<TD></TD>标签对定义了表格行中的一个单元格

在了解TEACHER.db和TEACHER.html文件格式的基础上，我们需要编写Shell脚本实现TEACHER.db到TEACHER.html的转换，该脚本可以通过三个步骤来完成：

} 建立HTML文件开始处的模板文件，直到<TABLE>之前；

} 将TEACHER.db的记录放到<TD></TD>标签对内；

} 建立HTML文件结束时的模板文件，从<\TABLE>到结束。

```
#例17-1：htmlconver.sh脚本将文本文件转换为HTML文件
#!/bin/bash
cat << CLOUD        # Here-document用法，CLOUD是分界符
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
 <HEAD>
  <TITLE>
  教授的信息
  </TITLE>
 </HEAD>
 <BODY>
  <TABLE>
CLOUD           #建立HTML文件开始处的模板文件完成
#将标准输入的内容用sed命令进行处理，sed命令有3个编辑选项
#第1个编辑选项将域分割符(:)替换成</TD><TD>
#第2个编辑选项在每行起始处加上<TR><TD>
#第3个编辑选项在每行结束处加上</TD></TR>
sed -e 's/:/<\/TD><TD>/g' -e 's/^/<TR><TD>/g' -e 's/$/<\/TD><\/TR>/g'

cat << CLOUD
 </TABLE>
 </BODY>
 </HTML>
CLOUD                   #建立HTML文件结束时的模板文件完成
```

查找文本中n个出现频率最高的单词

计算机科学中有一个著名的问题：写一个文本处理程序，查找文本中*n*个出现频率最高的单词，输出结果需要显示这些单词出现的次数，并按照次数从大到小排序

查找文本中*n*个出现频率最高的单词是比较复杂的，解决复杂问题的常用方法是将复杂问题切分成若干个简单子问题加以解决，我们可以将查找文本中*n*个出现频率最高的单词这一问题切分为如下几个子问题：

} 将文本文件中以一行一个单词的形式显示出来；

} 将单词中的大写字母转化为小写字母，即Word和word认作是一个单词；

} 对单词进行排序；

} 对排序好的单词列表统计每个单词出现的次数；

} 最后显示单词列表的前n项。

```
#例17-2：topn.sh脚本查找文本中n个出现频率最高的单词
#!/bin/bash
```

```
end=$1              #$1是输出频率最高单词的个数

cat $2 |            #$2是目标文本文件名称
tr -cs "[a-z][A-Z]" "[\012*]" |      #将文本文件中以一行一个单词的形式显示出来
  tr A-Z a-z |                  #将单词中的大写字母转化为小写字母
    sort |                    #对单词进行排序
     uniq -c |                #对排序好的单词列表统计每个单词出现的次数
       sort -k1nr -k2 |       #按出现频率排序，再按字母顺序排序
        head -n"$end"         #显示前n行
```

伪随机数的产生和应用

计算机不会产生绝对随机的随机数，计算机只能产生“伪随机数”

bash Shell同样也定义了伪随机数产生工具，即$RANDOM函数，每次调用这个函数将返回一个伪随机整数，范围在0-32767之间

随机数在互联网中具有广泛的应用背景，如计算机仿真模拟、数据加密、网络游戏等

} 实现图片验证的关键技术就是产生随机数，下面的脚本seqrand.sh用于生成一段定长的随机字符串，字符串由数字和大小写字母组成

} dice.sh脚本利用随机数实现掷骰子功能

```
#例17-4：seqrand.sh脚本产生定长的随机字符串
#!/bin/bash

length=6                    #随机字符串的长度
i=1                         #计数器，初值是1

#seq是数字、大小写字母的序列
seq=(0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)

num_seq=${#seq[@]}                  #计算seq数组的长度

while [ "$i" -le "$length" ]        #产生$length个随机字符
do
 seqrand[$i]=${seq[$((RANDOM%num_seq))]}    #取模产生num_seq内的随机数
 let "i=i+1"
done


#例17-5：dice.sh脚本模拟投骰子游戏
#!/bin/bash

PIPS=6                  #一个骰子有6面
MAX=1000                #投骰子的次数
throw=1                 #计数器，初值是1
```

```
#下面是6个计数的变量
one=0
two=0
three=0
four=0
five=0
six=0
count()                    #更新计数的次数
{
case "$1" in
 0) let "one=one+1";;            #跟$PIPS变量取模为0-5，因此，0代表骰子的1
 1) let "two=two+1";;
 2) let "three=three+1";;
 3) let "four=four+1";;
 4) let "five=five+1";;
 5) let "six=six+1";;
esac
}

while [ "$throw" -le "$MAX" ]          #开始掷骰子
do
  let "dice=RANDOM % $PIPS"       #dice是0-5之间的随机数
  count $dice                     #更新统计次数的值
  let "throw=throw+1"
done
```

crontab的设置和应用

crontab命令的功能是在一定的时间间隔调度一些命令的执行。在/etc目录下有一个crontab文件，这里存放有系统运行的一些调度程序

} crontab -u user -e

} vi /etc/crontab   #与上面两条命令是等价的

} crontab还有下表所示的几个选项

选项名称	意义
e	编辑crontab调度表
l	在stdout上显示crontab
r	删除当前的crontab文件

crontab中的语句格式为：

} * * * * * 用户名 可执行命令

} 前面的五个“*”分别表示“分钟、小时、日期、月份、星期”，取值范围分别为“0-59、0-23、1-31、1-12、0-6”

例17-7：某系统管理员需每天做一定的重复工作，请按照下列要求，编制一个解决方案：

} （1）在下午4 :50删除/abc目录下的全部子目录和全部文件；

} （2）从早8:00～下午6:00每小时读取/xyz目录下x1文件中每行第一个域的全部数据加入到/backup目录下的bak01.txt文件内；

} （3）每逢星期一下午5:50将/data目录下的所有目录和文件归档并压缩为文件：backup.tar.gz；

} （4）在下午5:55将IDE接口的CD-ROM卸载（假设：CD-ROM的设备名为hdc）；

} （5）在早晨8:00前开机后启动。

```
[root@zawu shell-program]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)  OR
#sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  *  command to be executed

50 16 * * * rm -r /abc/*                        #完成目标（1）
0 8-18/1 * * * cut -f1 /xyz/x1 >;>; /backup/bak01.txt    #完成目标（2）
50 17 * * * tar zcvf backup.tar.gz /data              #完成目标（3）
55 17 * * * umount /dev/hdc                     #完成目标（4）
```

使用MySQL数据库

数据库能永久存储各种数据，Shell脚本经常需要从数据库中读取数据、处理数据、再将结果写回到数据库

以MySQL为代表，介绍Shell脚本与DBMS的交互

Fedora Core 11系统，MySQL安装需要依次安装下面的包：

} perl-DBI-1.607-2.fc11.i586.rpm

} perl-DBD-MySQL-4.010-1.fc11.i586.rpm

} mysql-5.1.32-1.fc11.i586.rpm

} mysql-server-5.1.32-1.fc11.i586.rpm

登录数据库脚本

```
#例17-9：log.sh脚本登录MySQL服务器
#!/bin/bash

MYSQL=`which mysql`    #利用命令替换获得mysql的路径

$MYSQL -u mysql -p      #以mysql用户登录MySQL数据库
```

} 上述两条语句可根据不同DBMS而改变

```
PSQL=`which psql`
$PSQL test
```

} 数据库登录成功之后，就可以向DBMS发送SQL语句

} 建议使用两种方法：

} Here document方法

```
#例17-11：create.sh脚本创建people表
#!/bin/bash

MYSQL=`which mysql`
```

```
$MYSQL test -u mysql -p <<EOF            #Here-document用法
#创建people表，包含4个属性
create table people(name VARCHAR(20),sex CHAR(1),birth DATE,birthaddr VARCHAR(20));
show tables;
#可以加入任意MySQL命令和SQL语句
EOF
```

将SQL语句保存到Shell变量，再通过Here document方法提交到DBMS

```
#例17-13：insert2.sh脚本演示按照name sex birth birthaddr的顺序插入记录
#!/bin/bash
MYSQL=`which mysql`
if [ $# -ne 4 ]                    #输入参数不能少于4个
then
  echo "Usage:insert2.sh name sex birth birthaddr"
else
  statement="insert into people values ('$1','$2','$3','$4');"     #生成SQL语句
$MYSQL test -u mysql -p <<EOF
$statement                         #灵活的用法！
EOF
 if [ $? -eq 0 ]                   #判定退出码是否正常
 then
  echo "Data sucessfully added."
else
 echo "Problem adding data"
 fi
fi
```

Linux服务器性能监控系统

Linux服务器性能监控，就是在网络环境下为管理系统及终端用户提供性能参数信息、服务器状态等，所收集到的性能参数可以为管理系统制定决策、分配和调度资源等提供依据，并利于第一时间发现服务器故障，及时恢复调整服务器系统

Ganglia是一种可扩展的分布式监控系统，主要用于监控各种网络服务器、集群系统等高性能计算机系统

} 客户端Ganglia Monitoring Daemon (gmond)

} 服务端Ganglia Meta Daemon (gmetad)

} Ganglia PHP Web Frontend（基于web的动态访问方式）

参数名称	意义
cpu_idle	空闲CPU百分比
cpu_num	CPU数目，实际上是CPU核的数目
cpu_speed	CPU 的主频，单位为MHz
disk_free	总的磁盘空间
disk_total	空闲的磁盘空间
load_fifteen	15分钟内的CPU平均负载
load_five	5分钟内的CPU平均负载
load_one	1分钟内的CPU平均负载
machine_type	机器类型，一般为X_86_64
mem_buffers	缓冲区内存的大小
mem_cached	Cache的大小
mem_free	总的物理内存大小
mem_shared	总的虚拟内存大小
mem_total	总的内存大小，物理内存加上虚拟内存
mtu	网络最大的传输包的长度，单位是Byte
os_name	操作系统名字
os_release	操作系统的发行版本
swap_free	空闲的交换区的空间
swap_total	总的交换区空间

Ganglia安装，按Linux源码包的安装方式：

} tar zxvf ganglia-3.0.3.tar.gz

} cd ganglia-3.0.3

} ./configure

} make

} make install

[root@jselab local]# telnet localhost 8649 > gmond_msg_1.txt

[root@jselab local]# cat gmond_msg_1.txt

#以下一段是Ganglia出现的系统信息，与具体性能参数无关

#因而，提取性能参数时可以忽略这段信息

Trying 127.0.0.1...

Connected to localhost.localdomain (127.0.0.1).

Escape character is '^]'.

<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<!DOCTYPE GANGLIA_XML [

  <!ELEMENT GANGLIA_XML (GRID|CLUSTER|HOST)*>

#下面开始出现组播网段内所有服务器的性能参数名称及其数据

#因而，需要从下面格式的数据中提取出有用的信息

<HOST NAME="seugrid2.seu.edu.cn" IP="172.18.12.178" REPORTED="1228044273" TN="2" TMAX="20" DMAX="0" LOCATION="unspecified" GMOND_STARTED="1225956735">
<METRIC NAME="disk_total" VAL="25.618" TYPE="double" UNITS="GB" TN="3138" TMAX="1200" DMAX="0" SLOPE="both" SOURCE="gmond"/>
<METRIC NAME="cpu_speed" VAL="2992" TYPE="uint32" UNITS="MHz" TN="734" TMAX="1200" DMAX="0" SLOPE="zero" SOURCE="gmond"/>